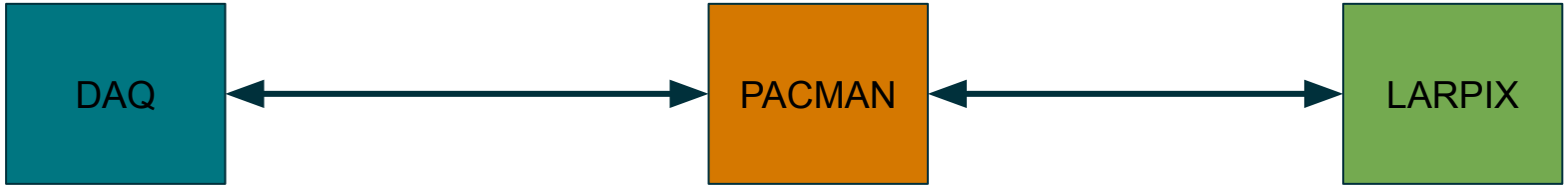
LArPix readout chain logic

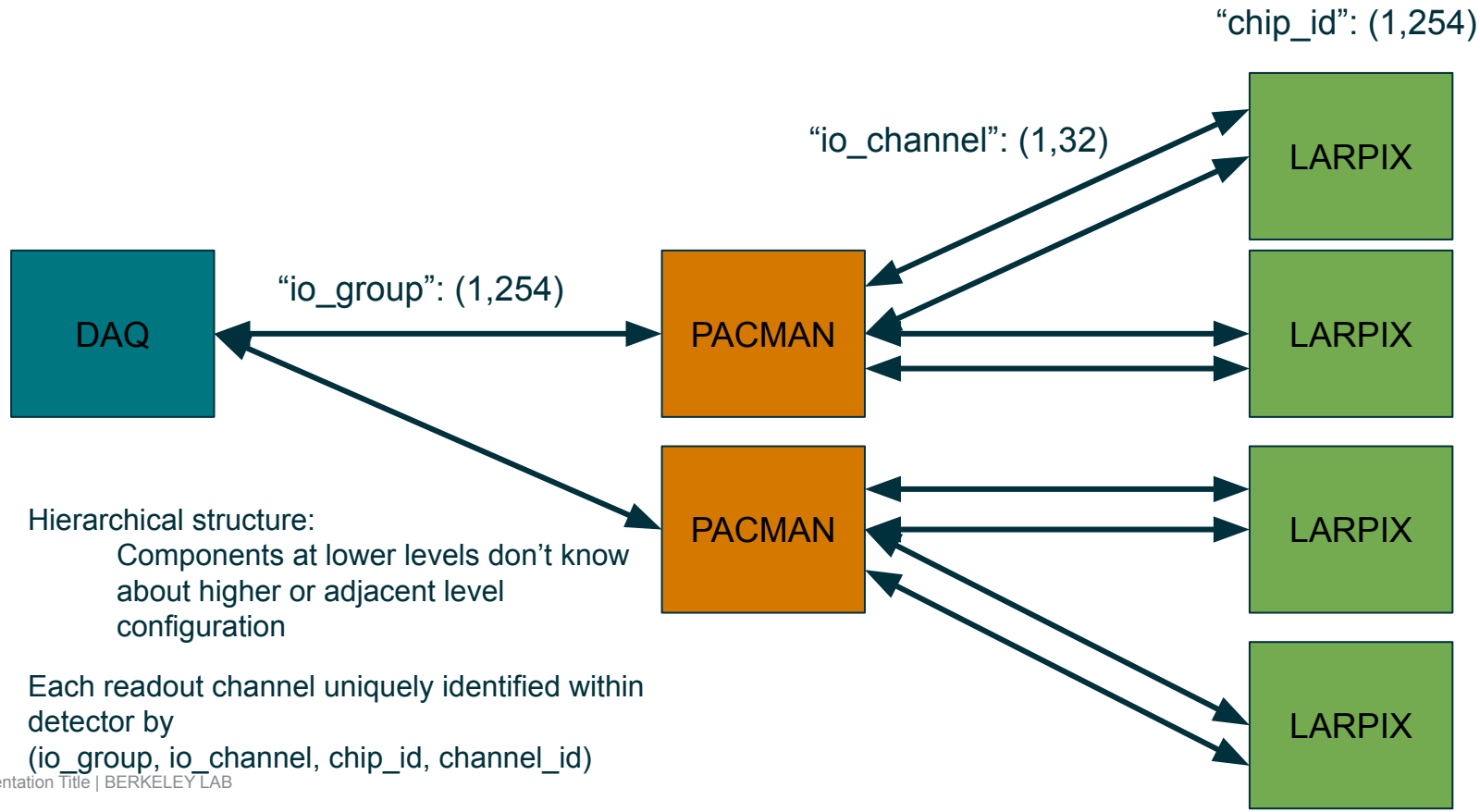
DUNE-LArPix meeting
2/18/21
Peter Madigan



3 components to the readout chain:

- LArPix (analog + digital)
- PACMAN (warm-side electronics)
- DAQ (central control + persistent data format)

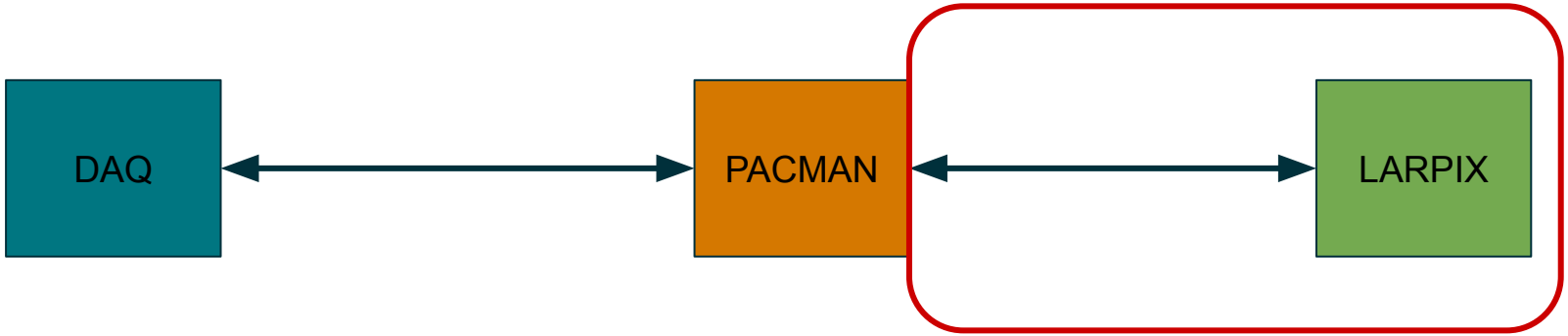
I'm going to give an overview of the main *logic* of each interface



Hierarchical structure:

Components at lower levels don’t know about higher or adjacent level configuration

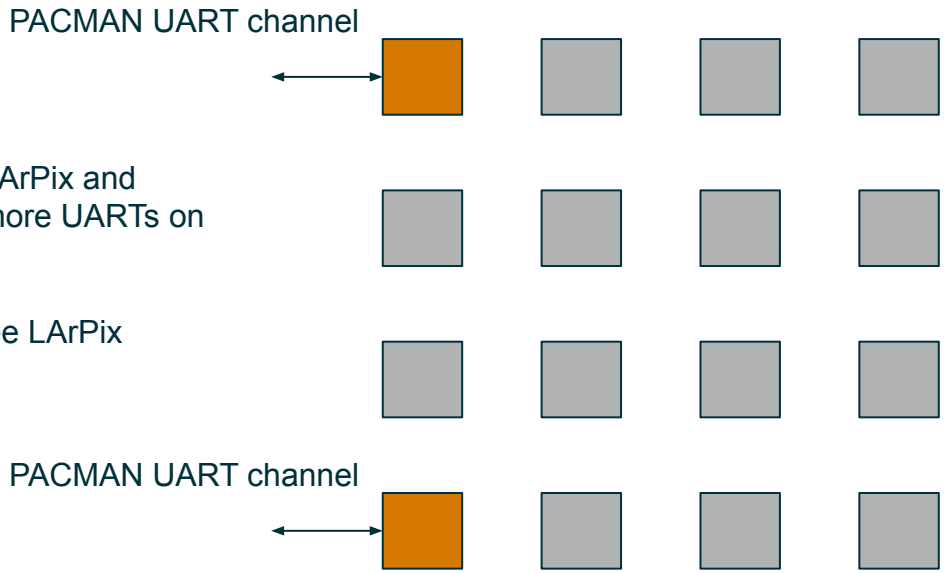
Each readout channel uniquely identified within detector by
(io_group, io_channel, chip_id, channel_id)


DAQ
PACMAN
LARPIX

LArPix tile interface

Communication between LArPix and PACMAN occurs via 1 or more UARTs on a select few chips.

UART format is custom (see LArPix datasheet for details).

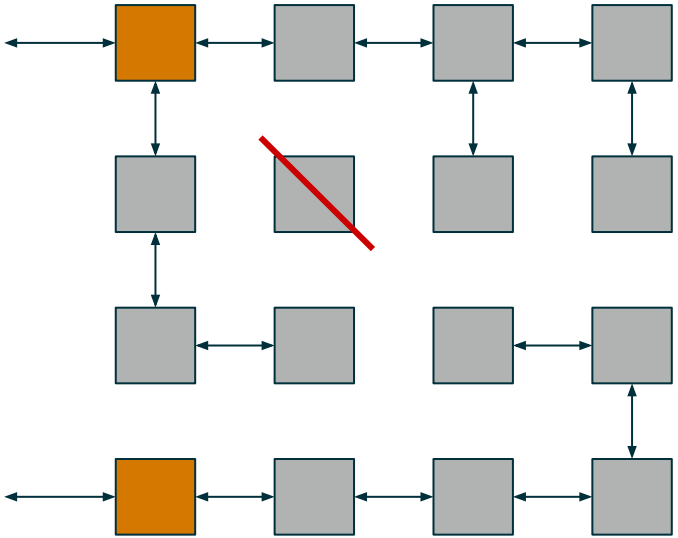


Hydra network

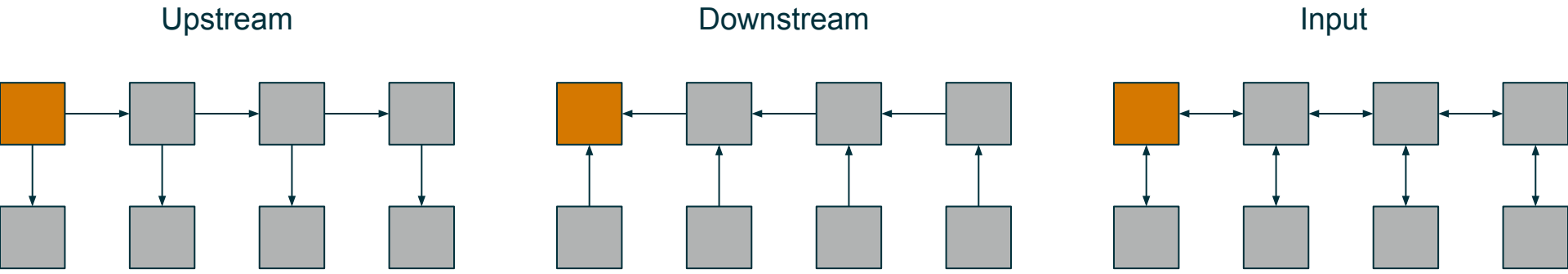
Each ASIC acts as a switch, forwarding data to neighbors.

Communication between chips can be configured to route around failed chips.

Network must be initialized to communicate with all chips.



Complete, arbitrary representation of hydra network

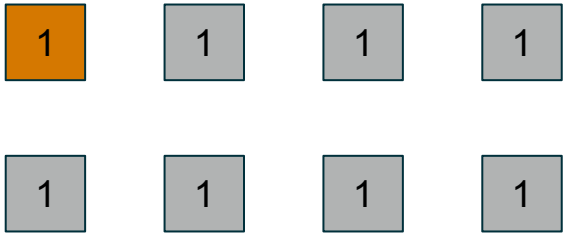


miso_us, miso_ds, mosi

Hydra network is determined by three configurations:

Upstream	: send config-write data (miso_us)
Downstream	: send data/config-read type data (miso_ds)
Input	: receive data (mosi)

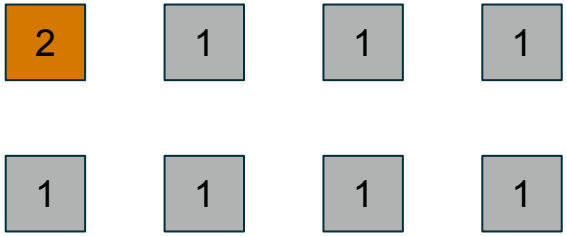
Hydra network initialization



Initial state:

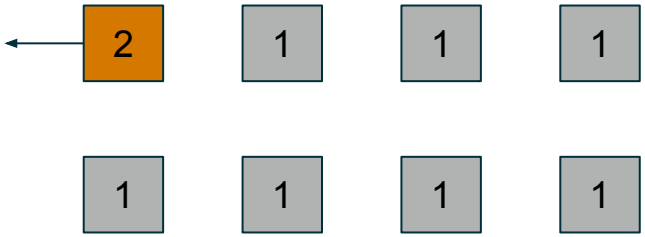
- All chips listening on all uarts
- No chips sending data
- Chip id = 1

Hydra network initialization



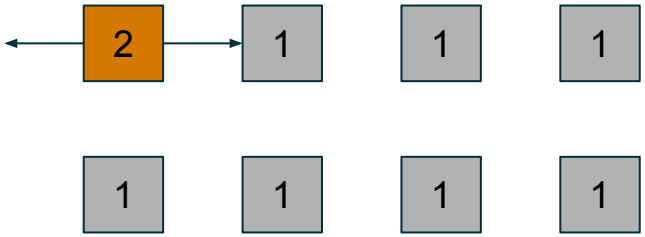
Set chip id (2)

Hydra network initialization



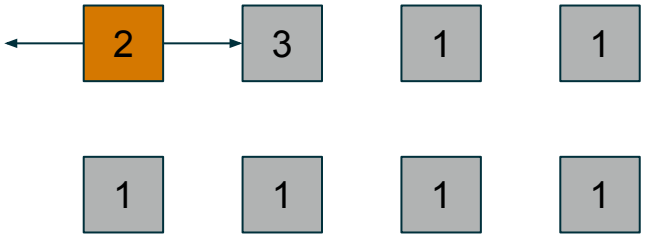
Set downstream (2)

Hydra network initialization



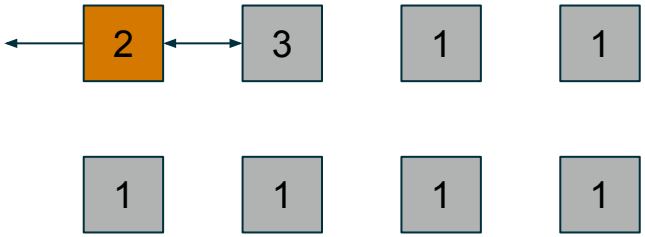
Set upstream (2)

Hydra network initialization



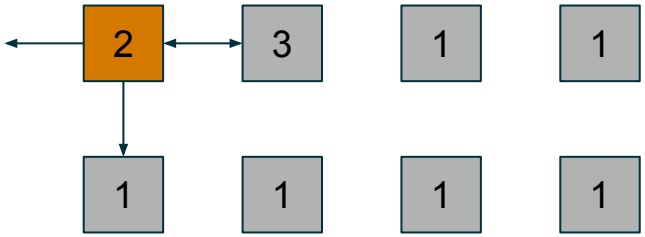
Set new chip id (3)

Hydra network initialization



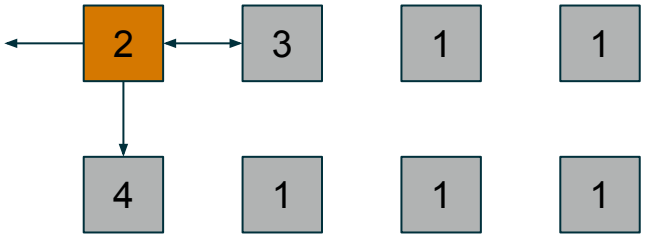
Set downstream (3)

Hydra network initialization



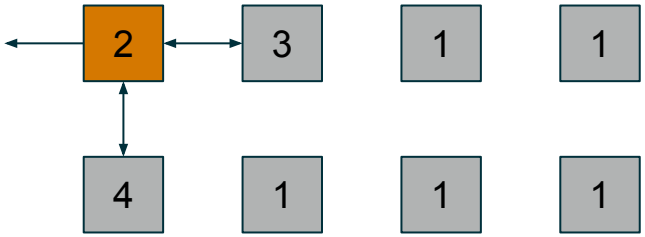
Set upstream (2)

Hydra network initialization



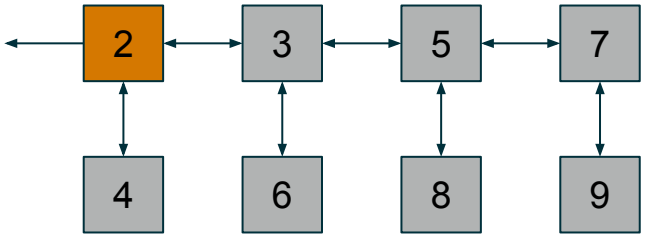
Set chip id (4)

Hydra network initialization



Set downstream (4)

Hydra network initialization



Yadda, yadda…

...very tedious

Within larpix-control

Automate it!

Controller.load(json_filepath)

Builds internal representation of desired hydra network (controller.network)

Controller.init_network(io_group, io_channel, chip_id=None)

Traverses hydra network, initializing each chip

Network config file

I will explicitly go over this

Hydra network config files

“_config_type”: “controller”
Declares what kind of a json config file this is

```
{
    "_config_type": "controller",
    "_include": [<optional list of files to inherit from>],
    "name": <string identifier for configuration ID, e.g. "pcb-3", or "acube-2x2">,
    "layout": <string identifier for layout version, e.g. "1.2.0">,
    "asic_version": 2,
    "network": {
        "miso_uart_map": [<uart channel for position in array, required>, ...],
        "mosi_uart_map": [<uart channel for position in array, required>, ...],
        "usds_link_map": [<position of chip relative to neighbor, required>, ...],
        "<io_group>": {
            "miso_uart_map": [<uart channels for io_group, optional>, ...],
            ...
            "<io_channel>": {
                "miso_uart_map": [<uart channels for io_group, optional>, ...],
                ...
                "nodes": [
                    {
                        "chip_id": <chip id of first chip to load into network>,
                        "miso_us": [<chip_ids of upstream chips, null if no upstream chip,
                        "miso_ds": [<chip_ids of downstream chips, optional>],
                        "mosi": [<chip_ids of chips to listen to, optional>],
                        "root": <bool indicating if this is a root node, optional>,
                        "miso_uart_map": [<uart channels for chip, optional>, ...],
                        ...
                    },
                    ...
                ]
            },
            ...
        },
        ...
    }
}
```

Hydra network config files

"_include": []
Specify other json files to include in config

Can write config files for single tile(s) and combine in a "master config"

io_group / io_channel must be unique in each file

(Future upgrade?: Use jsonnet/yaml to make this cleaner)

```
{
    "_config_type": "controller",
    "_include": [<optional list of files to inherit from>],
    "name": <string identifier for configuration ID, e.g. "pcb-3", or "acube-2x2">,
    "layout": <string identifier for layout version, e.g. "1.2.0">,
    "asic_version": 2,
    "network": {
        "miso_uart_map": [<uart channel for position in array, required>, ...],
        "mosi_uart_map": [<uart channel for position in array, required>, ...],
        "usds_link_map": [<position of chip relative to neighbor, required>, ...],
        "<io_group>": {
            "miso_uart_map": [<uart channels for io_group, optional>, ...],
            ...
            "<io_channel>": {
                "miso_uart_map": [<uart channels for io_group, optional>, ...],
                ...
                "nodes": [
                    {
                        "chip_id": <chip id of first chip to load into network>,
                        "miso_us": [<chip_ids of upstream chips, null if no upstream chip,
                        "miso_ds": [<chip_ids of downstream chips, optional>],
                        "mosi": [<chip_ids of chips to listen to, optional>],
                        "root": <bool indicating if this is a root node, optional>,
                        "miso_uart_map": [<uart channels for chip, optional>, ...],
                        ...
                    },
                    ...
                ]
            },
            ...
        },
        ...
    }
}
```

Hydra network config files

“name”: “”
Additional identifier, but not used by anything

“layout”: “”
Nominally associated with a larpix-geometry pixel map, but not used by anything internal to larpix-control

```
{
    "_config_type": "controller",
    "_include": [<optional list of files to inherit from>],
    "name": <string identifier for configuration ID, e.g. "pcb-3", or "acube-2x2">,
    "layout": <string identifier for layout version, e.g. "1.2.0">,
    "asic_version": 2,
    "network": {
        "miso_uart_map": [<uart channel for position in array, required>, ...],
        "mosi_uart_map": [<uart channel for position in array, required>, ...],
        "usds_link_map": [<position of chip relative to neighbor, required>, ...],
        "<io_group>": {
            "miso_uart_map": [<uart channels for io_group, optional>, ...],
            ...
            "<io_channel>": {
                "miso_uart_map": [<uart channels for io_group, optional>, ...],
                ...
                "nodes": [
                    {
                        "chip_id": <chip id of first chip to load into network>,
                        "miso_us": [<chip_ids of upstream chips, null if no upstream chip,
                        "miso_ds": [<chip_ids of downstream chips, optional>],
                        "mosi": [<chip_ids of chips to listen to, optional>],
                        "root": <bool indicating if this is a root node, optional>,
                        "miso_uart_map": [<uart channels for chip, optional>, ...],
                        ...
                    },
                    ...
                ]
            },
            ...
        },
        ...
    }
}
```

Hydra network config files

“asic_version”: 2
Used for compatibility with larpix-v1 asic, larpix-v2 asic, lightpix asic, ...

```
{
    "_config_type": "controller",
    "_include": [<optional list of files to inherit from>],
    "name": <string identifier for configuration ID, e.g. "pcb-3", or "acube-2x2">,
    "layout": <string identifier for layout version, e.g. "1.2.0">,
    "asic_version": 2,
    "network": {
        "miso_uart_map": [<uart channel for position in array, required>, ...],
        "mosi_uart_map": [<uart channel for position in array, required>, ...],
        "usds_link_map": [<position of chip relative to neighbor, required>, ...],
        "<io_group>": {
            "miso_uart_map": [<uart channels for io_group, optional>, ...],
            ...
            "<io_channel>": {
                "miso_uart_map": [<uart channels for io_group, optional>, ...],
                ...
                "nodes": [
                    {
                        "chip_id": <chip id of first chip to load into network>,
                        "miso_us": [<chip_ids of upstream chips, null if no upstream chip,
                        "miso_ds": [<chip_ids of downstream chips, optional>],
                        "mosi": [<chip_ids of chips to listen to, optional>],
                        "root": <bool indicating if this is a root node, optional>,
                        "miso_uart_map": [<uart channels for chip, optional>, ...],
                        ...
                    },
                    ...
                ]
            },
            ...
        },
        ...
    }
}
```

Hydra network config files

“network”: Hierarchy to describe full network structure

network[“1”] : description of io group 1’s network

network[“1”][“2”] : description of io channel 2’s network

network[“1”][“2”][“nodes”] : list of all “devices” on given hydra network

```
{
    "_config_type": "controller",
    "_include": [<optional list of files to inherit from>],
    "name": <string identifier for configuration ID, e.g. "pcb-3", or "acube-2x2">,
    "layout": <string identifier for layout version, e.g. "1.2.0">,
    "asic_version": 2,
    "network": {
        "miso_uart_map": [<uart channel for position in array, required>, ...],
        "mosi_uart_map": [<uart channel for position in array, required>, ...],
        "usds_link_map": [<position of chip relative to neighbor, required>, ...],
        "<io_group>": {
            "miso_uart_map": [<uart channels for io_group, optional>, ...],
            ...
            "<io_channel>": {
                "miso_uart_map": [<uart channels for io_group, optional>, ...],
                ...
                "nodes": [
                    {
                        "chip_id": <chip id of first chip to load into network>,
                        "miso_us": [<chip_ids of upstream chips, null if no upstream chip,
                        "miso_ds": [<chip_ids of downstream chips, optional>],
                        "mosi": [<chip_ids of chips to listen to, optional>],
                        "root": <bool indicating if this is a root node, optional>,
                        "miso_uart_map": [<uart channels for chip, optional>, ...],
                        ...
                    },
                    ...
                ]
            },
            ...
        },
        ...
    }
}
```

Hydra network config files

Each node:
“chip_id” : int for larpix asics, string for “other”
“miso_us” : list of chip_ids that this node sends upstream data to
“miso_ds” : list of chip_ids that this node sends downstream data to
“mosi” : list of chip_ids that this node receives data from
“root” : indicates start point for network traversal

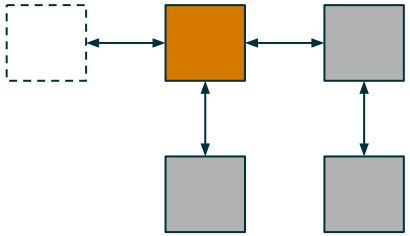
“miso_ds” and “mosi” are optional, if you are doing something simple (99.999% of the time)

One “root” node is needed per hydra network

```
{
    "_config_type": "controller",
    "_include": [<optional list of files to inherit from>],
    "name": <string identifier for configuration ID, e.g. "pcb-3", or "acube-2x2">,
    "layout": <string identifier for layout version, e.g. "1.2.0">,
    "asic_version": 2,
    "network": {
        "miso_uart_map": [<uart channel for position in array, required>, ...],
        "mosi_uart_map": [<uart channel for position in array, required>, ...],
        "usds_link_map": [<position of chip relative to neighbor, required>, ...],
        "<io_group>": {
            "miso_uart_map": [<uart channels for io_group, optional>, ...],
            ...
            "<io_channel>": {
                "miso_uart_map": [<uart channels for io_group, optional>, ...],
                ...
                "nodes": [
                    {
                        "chip_id": <chip id of first chip to load into network>,
                        "miso_us": [<chip_ids of upstream chips, null if no upstream chip,
                        "miso_ds": [<chip_ids of downstream chips, optional>],
                        "mosi": [<chip_ids of chips to listen to, optional>],
                        "root": <bool indicating if this is a root node, optional>,
                        "miso_uart_map": [<uart channels for chip, optional>, ...],
                        ...
                    },
                    ...
                ]
            },
            ...
        },
        ...
    }
}
```

Hydra network config files

Each node:
“Other” chips ids can (and should) be specified for placeholders, e.g. PACMAN:



Typically, *this* is your “root” node (“root”: true)

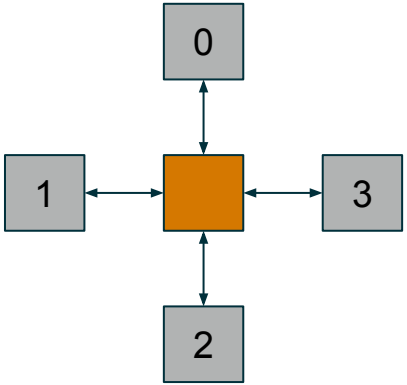
```
{
    "_config_type": "controller",
    "_include": [<optional list of files to inherit from>],
    "name": <string identifier for configuration ID, e.g. "pcb-3", or "acube-2x2">,
    "layout": <string identifier for layout version, e.g. "1.2.0">,
    "asic_version": 2,
    "network": {
        "miso_uart_map": [<uart channel for position in array, required>, ...],
        "mosi_uart_map": [<uart channel for position in array, required>, ...],
        "usds_link_map": [<position of chip relative to neighbor, required>, ...],
        "<io_group>": {
            "miso_uart_map": [<uart channels for io_group, optional>, ...],
            ...
            "<io_channel>": {
                "miso_uart_map": [<uart channels for io_group, optional>, ...],
                ...
                "nodes": [
                    {
                        "chip_id": <chip id of first chip to load into network>,
                        "miso_us": [<chip_ids of upstream chips, null if no upstream chip,
                        "miso_ds": [<chip_ids of downstream chips, optional>],
                        "mosi": [<chip_ids of chips to listen to, optional>],
                        "root": <bool indicating if this is a root node, optional>,
                        "miso_uart_map": [<uart channels for chip, optional>, ...],
                        ...
                    },
                    ...
                ]
            },
            ...
        },
        ...
    }
}
```

Hydra network config files

So far no knowledge of “geometry”

Defined by:
miso_uart_map, mosi_uart_map, usds_link_map

Each a list of 4 integers
Position in list is arbitrary, but must be *consistent*
Recommend the following:



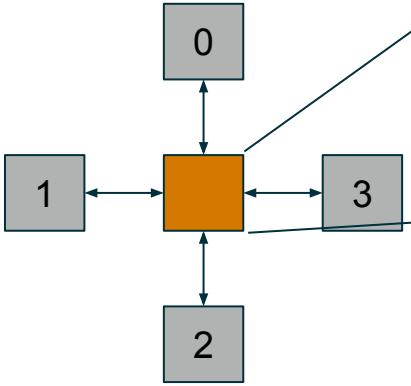

```
{
    "_config_type": "controller",
    "_include": [<optional list of files to inherit from>],
    "name": <string identifier for configuration ID, e.g. "pcb-3", or "acube-2x2">,
    "layout": <string identifier for layout version, e.g. "1.2.0">,
    "asic_version": 2,
    "network": {
        "miso_uart_map": [<uart channel for position in array, required>, ...],
        "mosi_uart_map": [<uart channel for position in array, required>, ...],
        "usds_link_map": [<position of chip relative to neighbor, required>, ...],
        "<io_group>": {
            "miso_uart_map": [<uart channels for io_group, optional>, ...],
            ...
            "<io_channel>": {
                "miso_uart_map": [<uart channels for io_group, optional>, ...],
                ...
                "nodes": [
                    {
                        "chip_id": <chip id of first chip to load into network>,
                        "miso_us": [<chip_ids of upstream chips, null if no upstream chip,
                        "miso_ds": [<chip_ids of downstream chips, optional>],
                        "mosi": [<chip_ids of chips to listen to, optional>],
                        "root": <bool indicating if this is a root node, optional>,
                        "miso_uart_map": [<uart channels for chip, optional>, ...],
                        ...
                    },
                    ...
                ]
            },
            ...
        },
        ...
    }
}
```

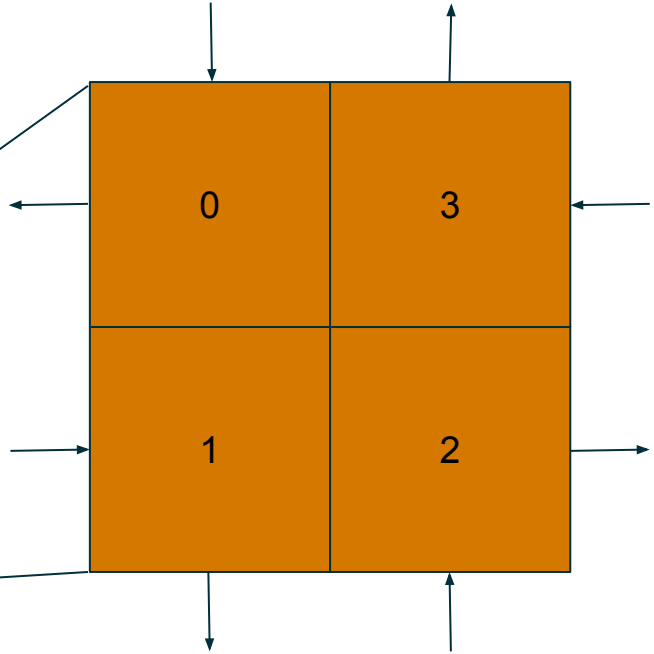
Hydra network config files

miso_uart_map:
UART number used to talk to chip at a given position

mosi_uart_map:
UART number used to listen to chip at a given position

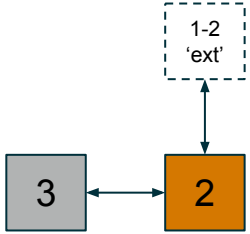
usds_link_map:
Map index of current chip *from perspective of other chip*



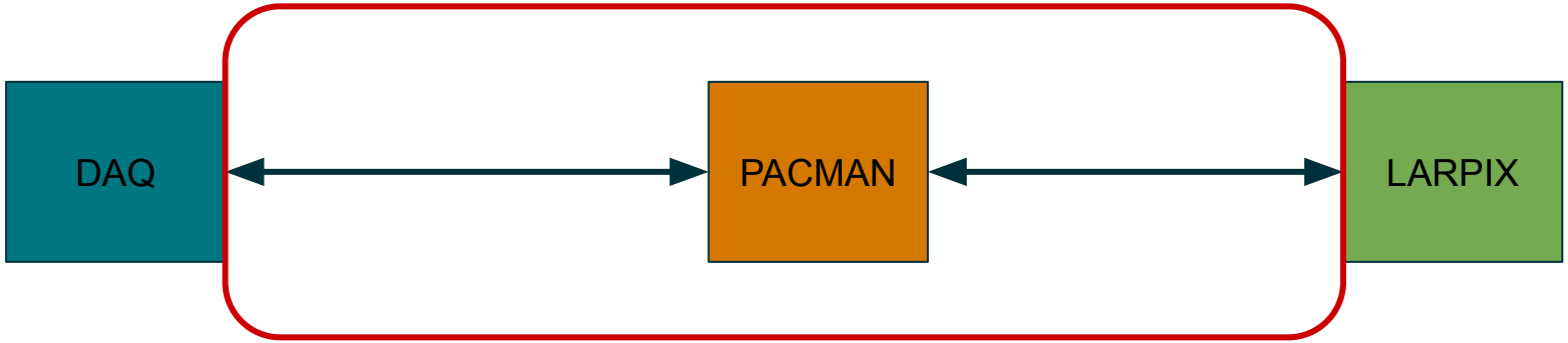


Example: 0 via quadrants 0 and 3
miso_uart_map[0] = 3
mosi_uart_map[0] = 0
usds_link_map[0] = 2

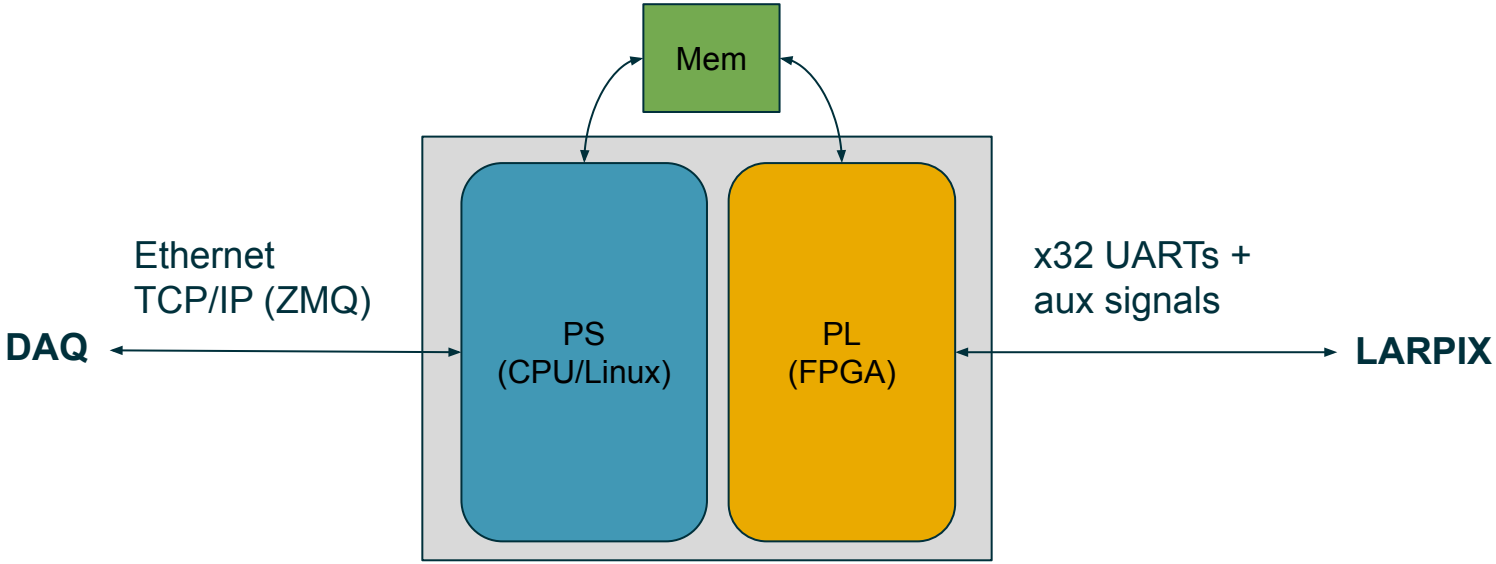
Hydra network config files




```
"network": {
    "1": {
        "2": {
            "nodes": [
                {
                    "chip_id": "ext",
                    "miso_us": [null,null,2,null],
                    "root": true
                }
                {
                    "chip_id": 2,
                    "miso_us": [null,3,null,null]
                },
                {
                    "chip_id": 3
                }
            ]
        }
    }
    "miso_uart_map": [3,0,1,2],
    "mosi_uart_map": [0,1,2,3],
    "usds_link_map": [2,3,0,1]
}
```

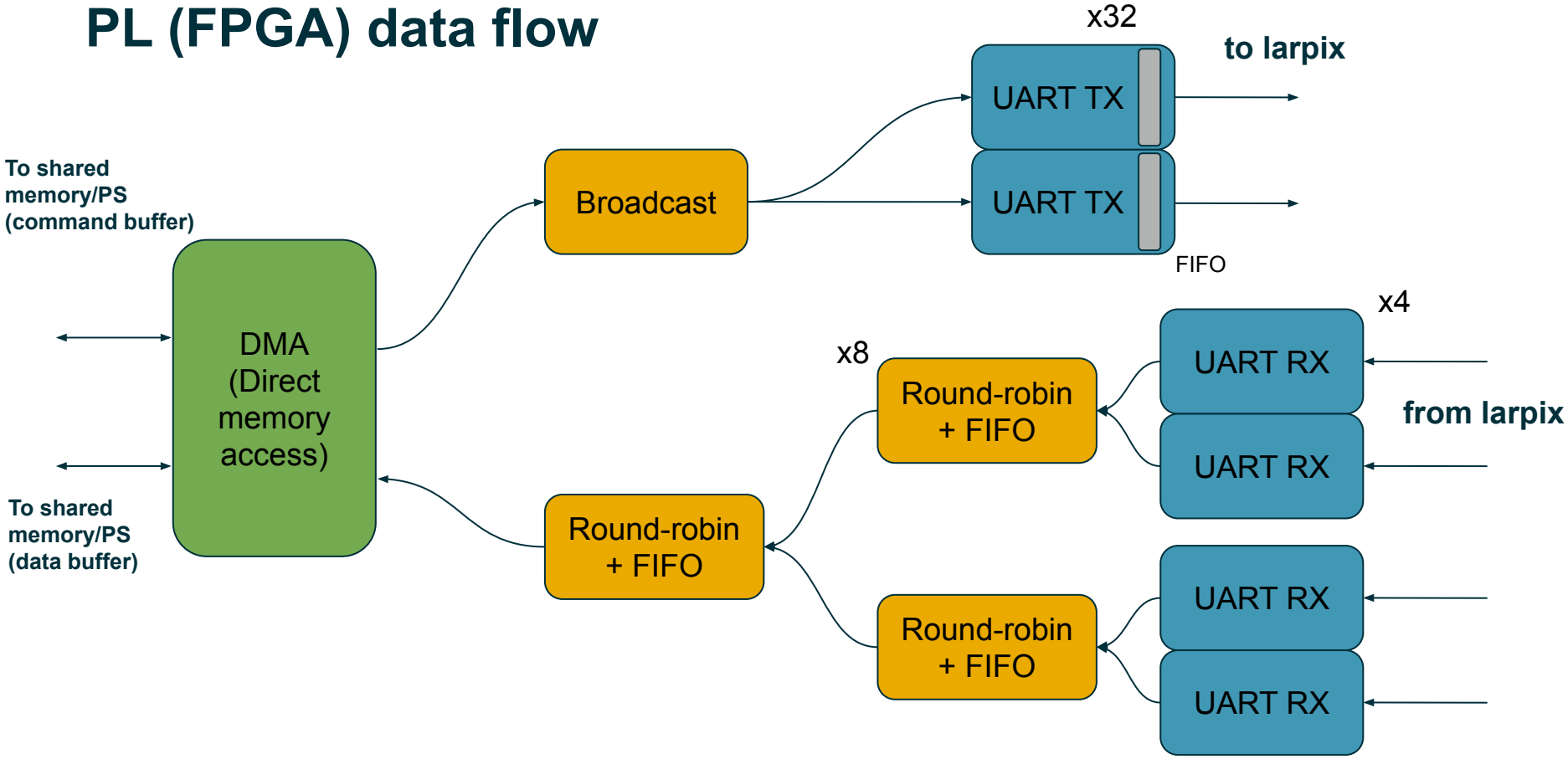

DAQ
PACMAN
LARPIX

PACMAN “firmware” overview

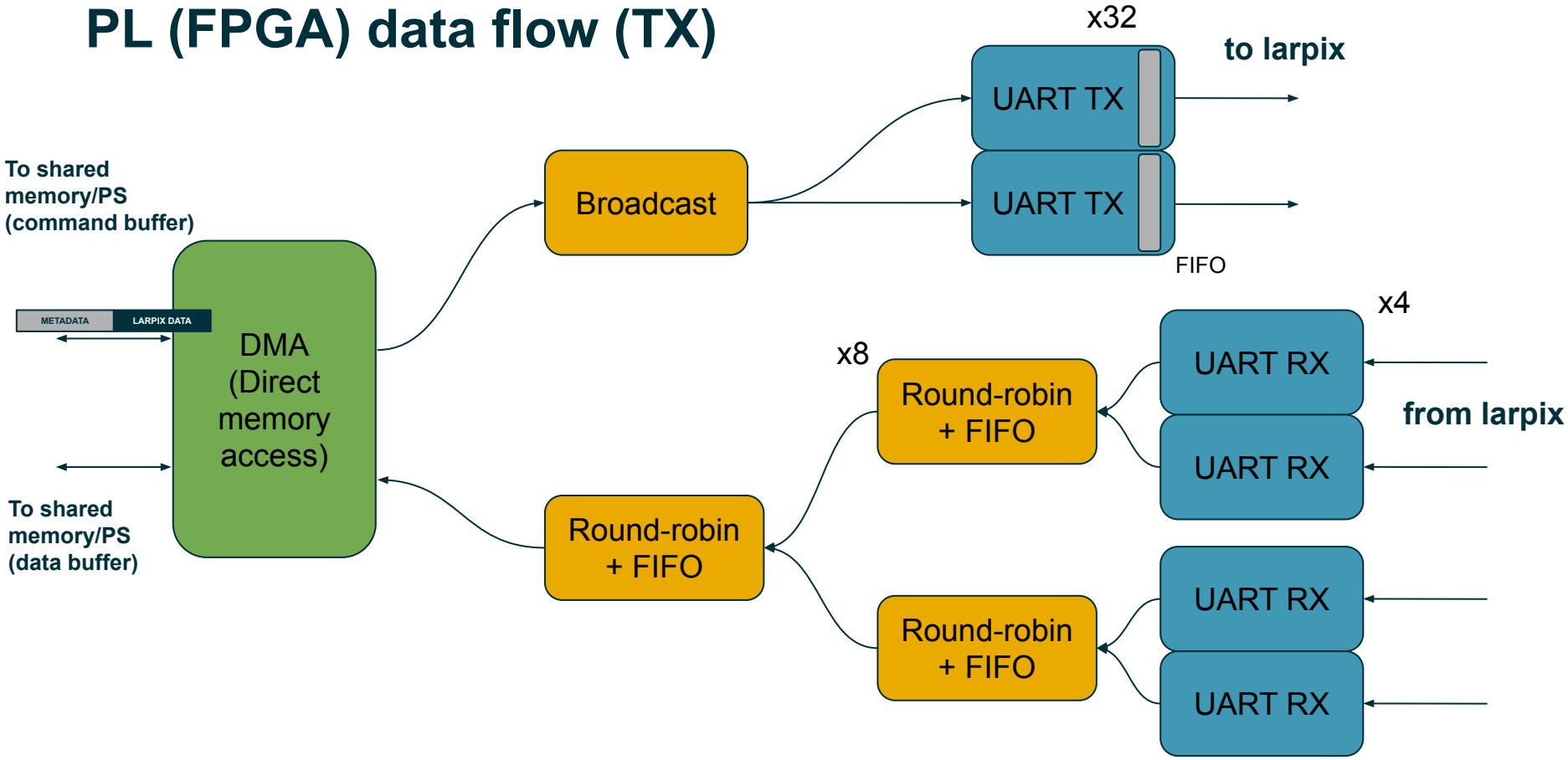


https://zeromq.org/ - Open source asynchronous messaging library
multi-language support, stable, well documented

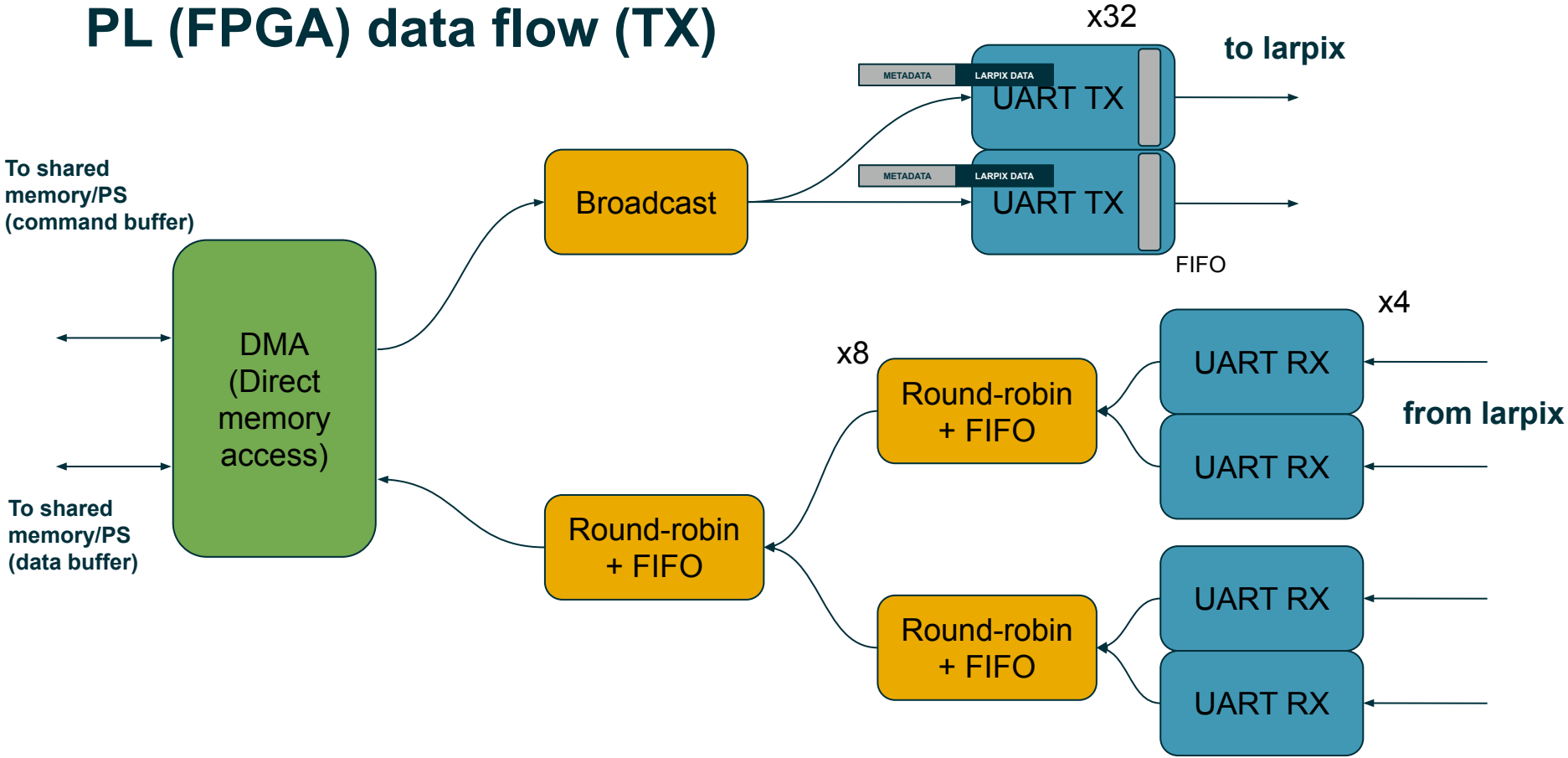
PL (FPGA) data flow



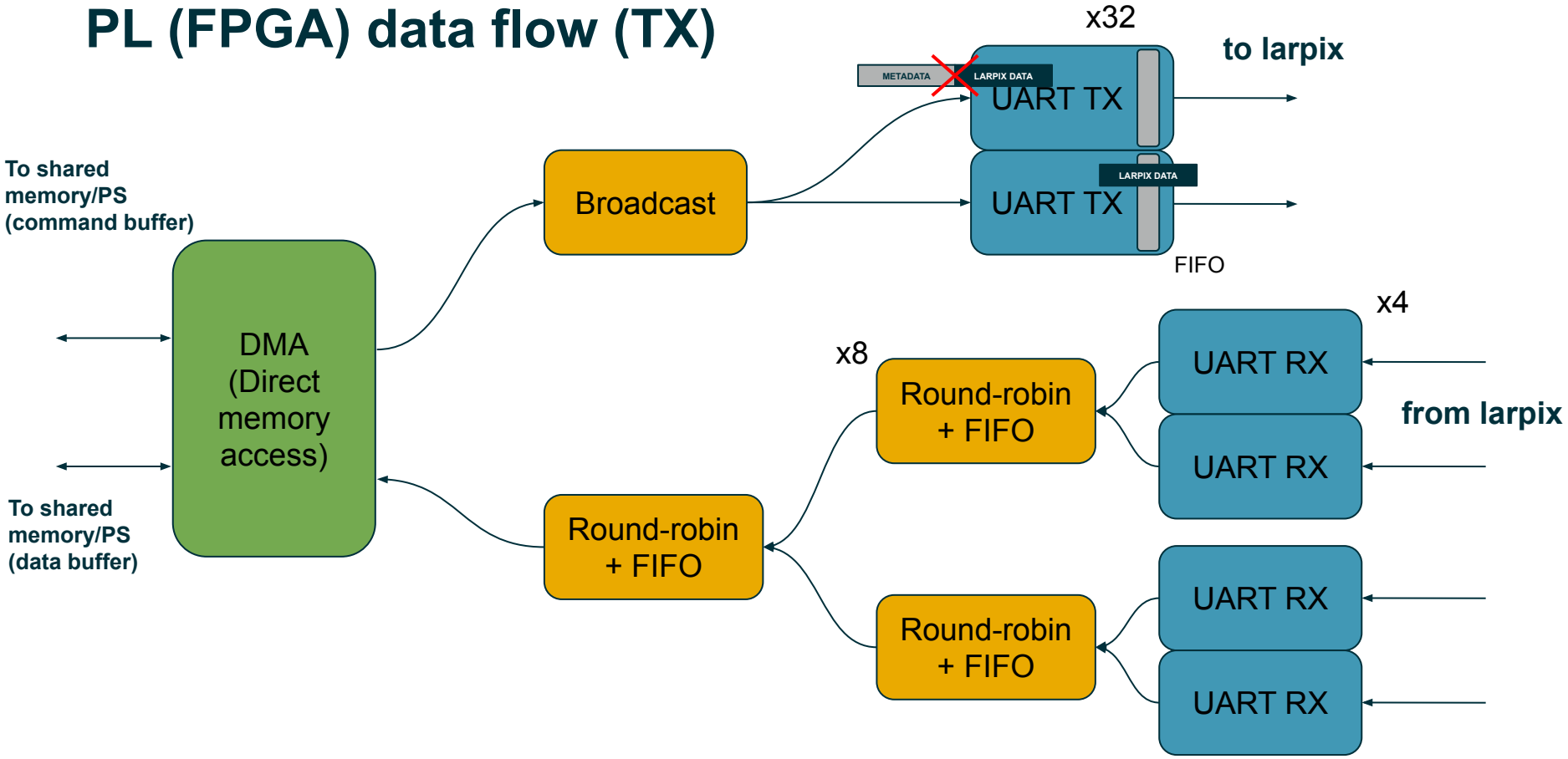
PL (FPGA) data flow (TX)



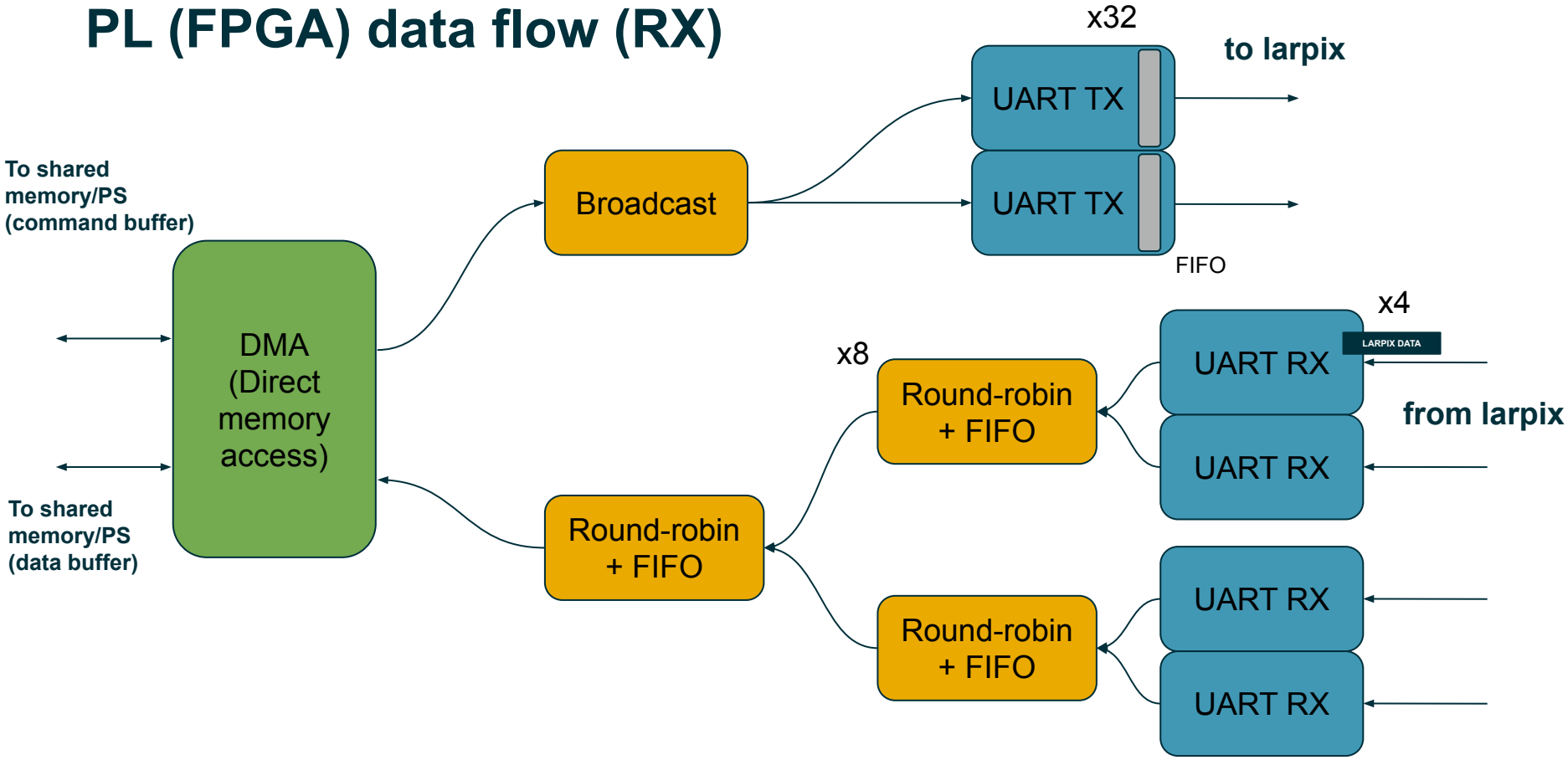
PL (FPGA) data flow (TX)



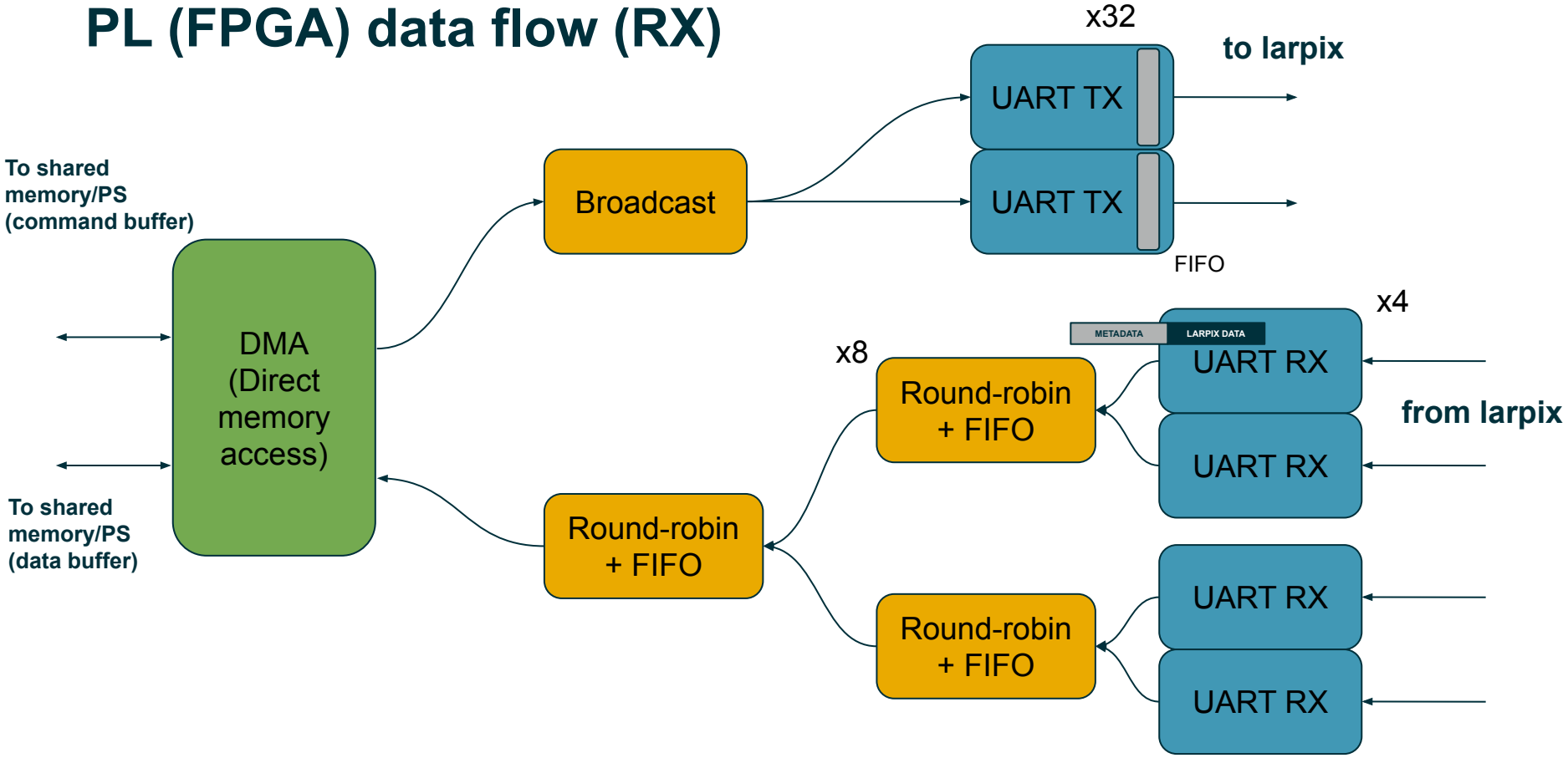
PL (FPGA) data flow (TX)



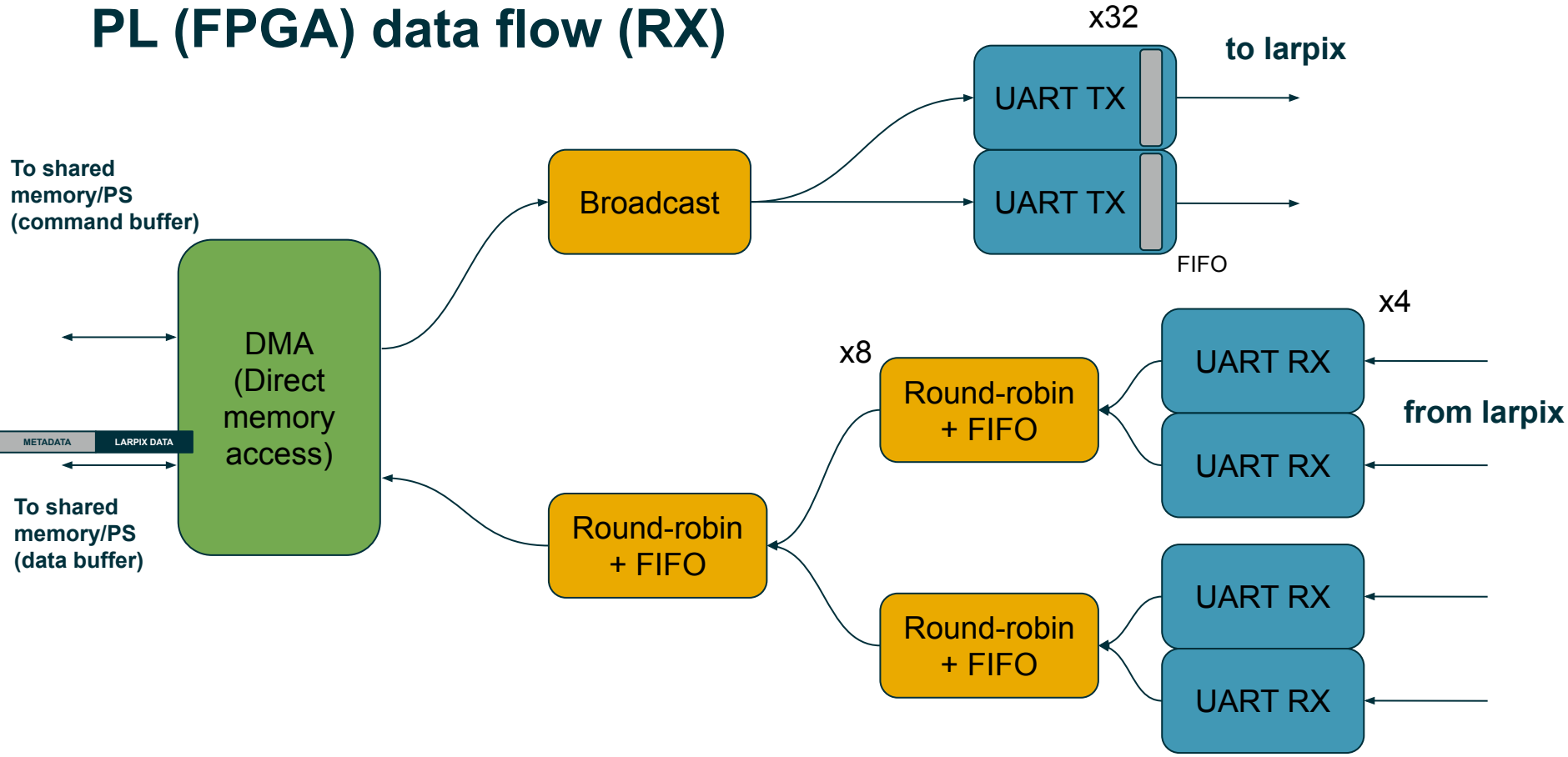
PL (FPGA) data flow (RX)



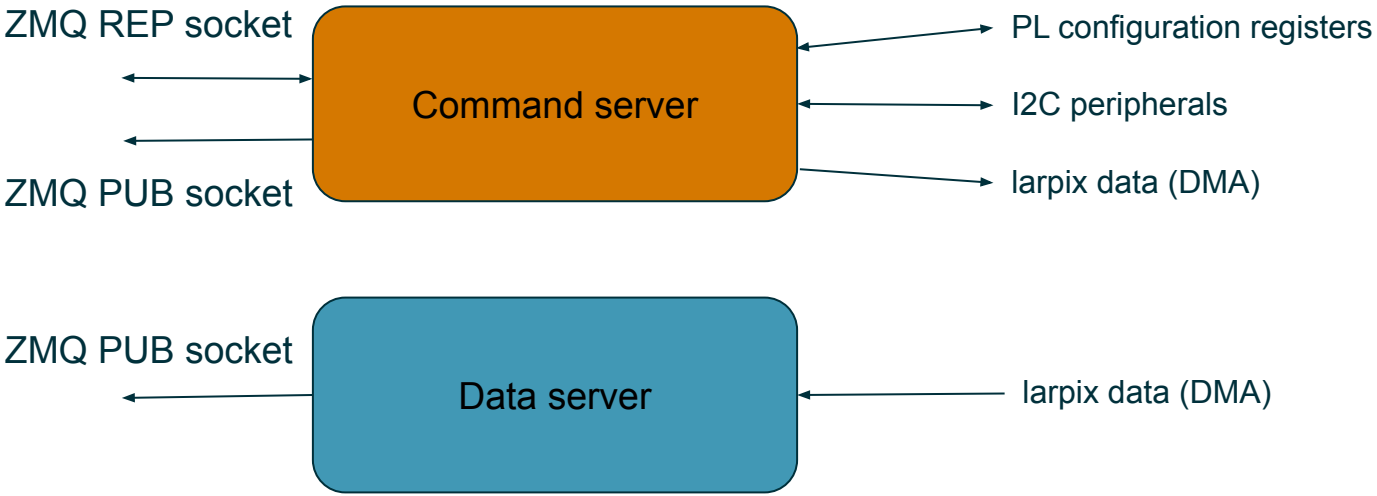
PL (FPGA) data flow (RX)



PL (FPGA) data flow (RX)



PS (Linux) servers



PACMAN servers source code: link

PACMAN interfacing doc: link

ZMQ message format

Multi-word format: 8-byte header, 16-byte words

Header: UNIX timestamp, word count, message type

Message types: data (from data-server), request (to command-server), reply (from command-server)

Words: type, word content

Word types: data (pass-through for larpix data), trigger, sync (or clock), read, write, ping, and error

Up to 65k words per message

See interfacing doc for specifics

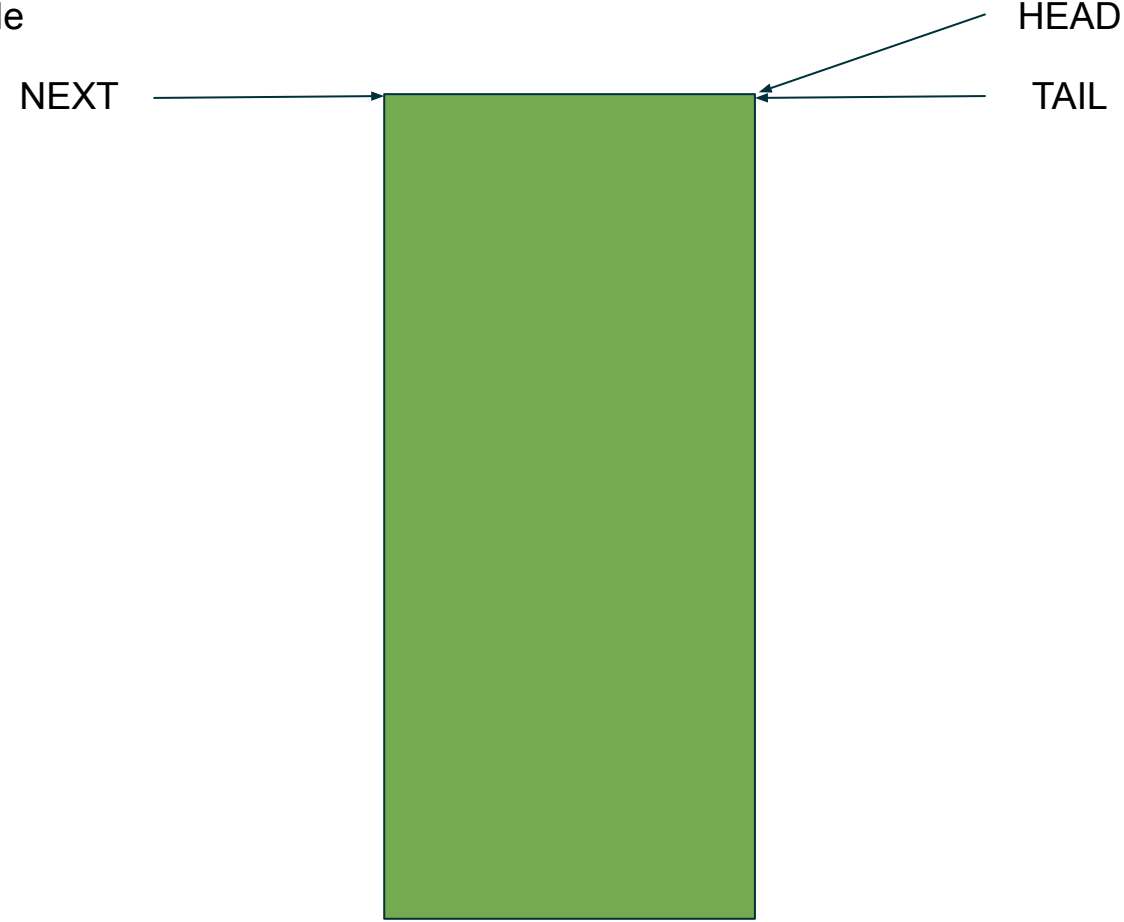
Command server

Pseudocode:

1. Wait for message
2. Loop over message words
    a. Do message “action”
        i. E.g. for transmit data: copy word to shared memory
3. If data in shared memory buffer
    a. Initiate DMA data transfer
    b. Wait for transfer complete
4. Respond to message (and “echo” response on PUB)
5. Go to 1.

Command server-side

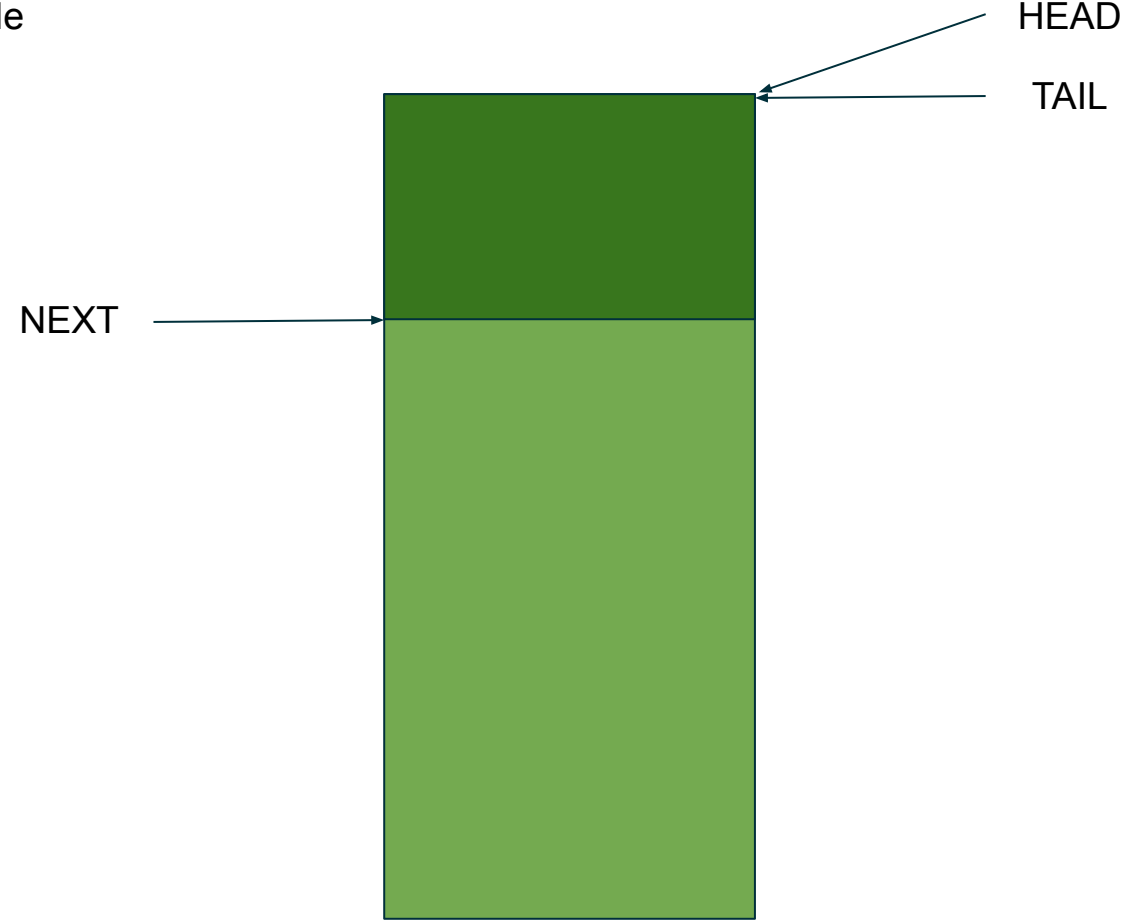
FPGA-side



Initial state

Command server-side

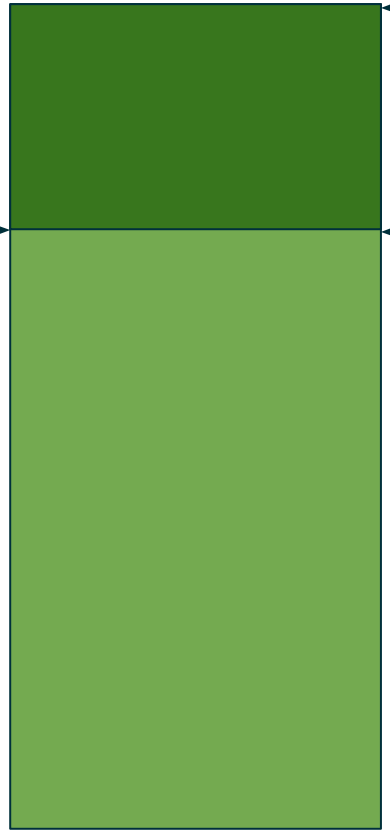
FPGA-side



Command server puts data into buffer

Command server-side

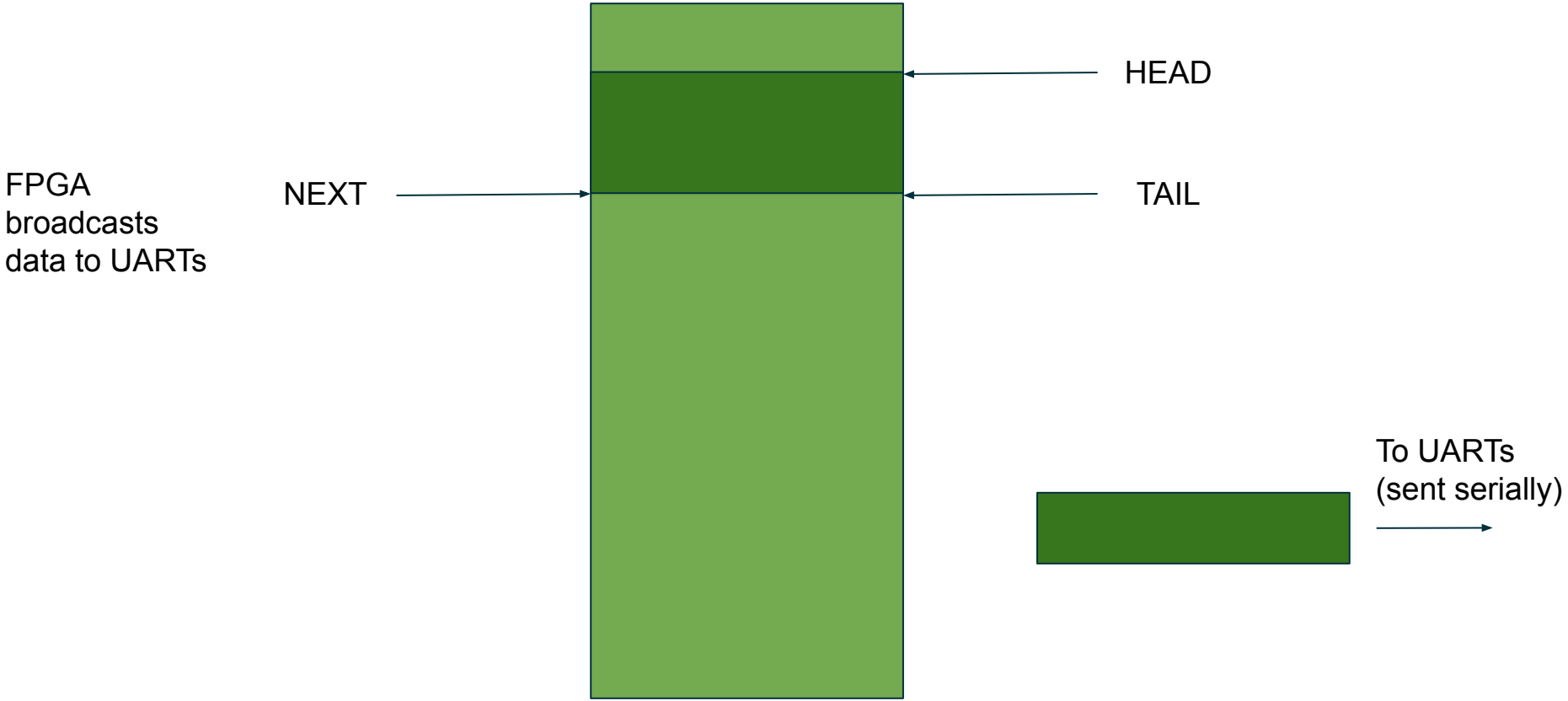
FPGA-side



Command updates TAIL pointer

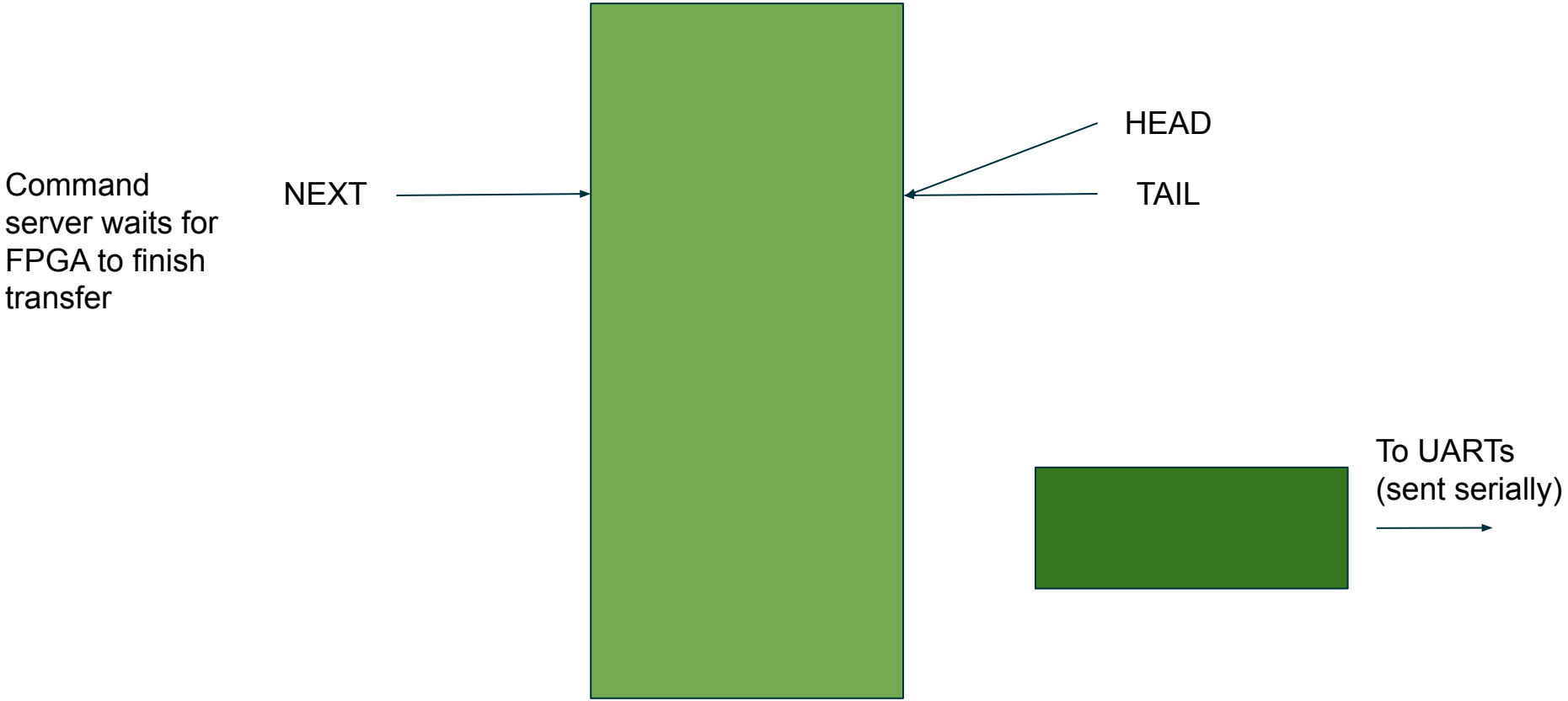
Command server-side

FPGA-side



Command server-side

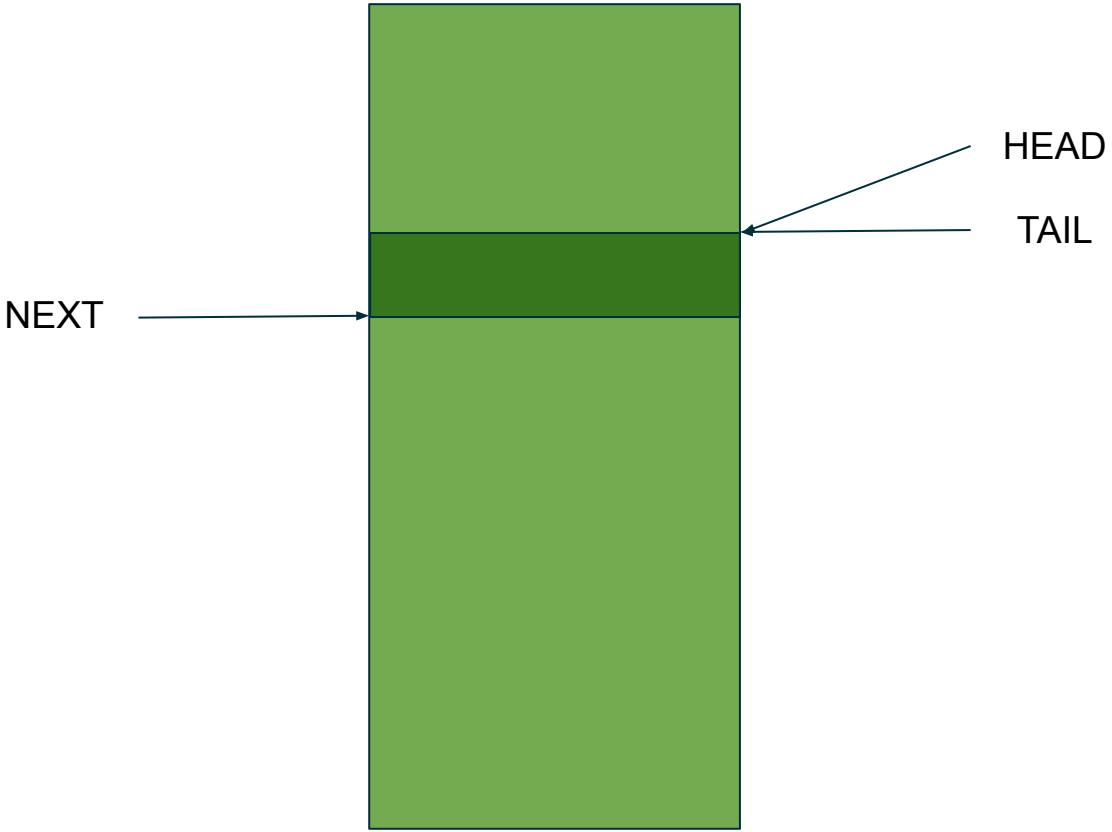
FPGA-side



Command server-side

FPGA-side

Repeat



Data server

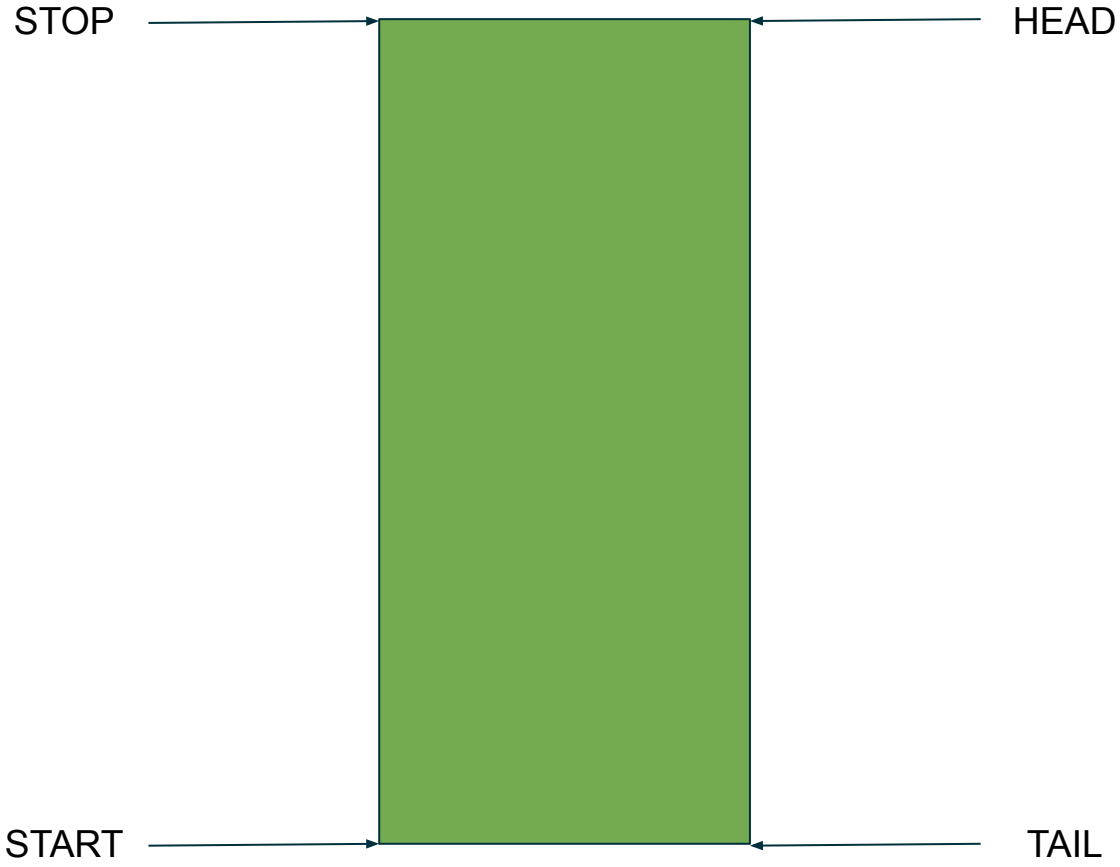
Pseudocode:

1. Wait for data in shared memory
2. Increment dataserver-side stop pointer to end of new data (or +65k words)
3. Wait for existing ZMQ message transfer to complete (1s timeout)
4. Loop over data between dataserver-side start and stop pointers
    a. Copy data to ZMQ message
    b. Increment start pointer
5. Update DMA tail to stop pointer
6. Pass message to ZMQ
7. Go to 1.

Dataserver-side

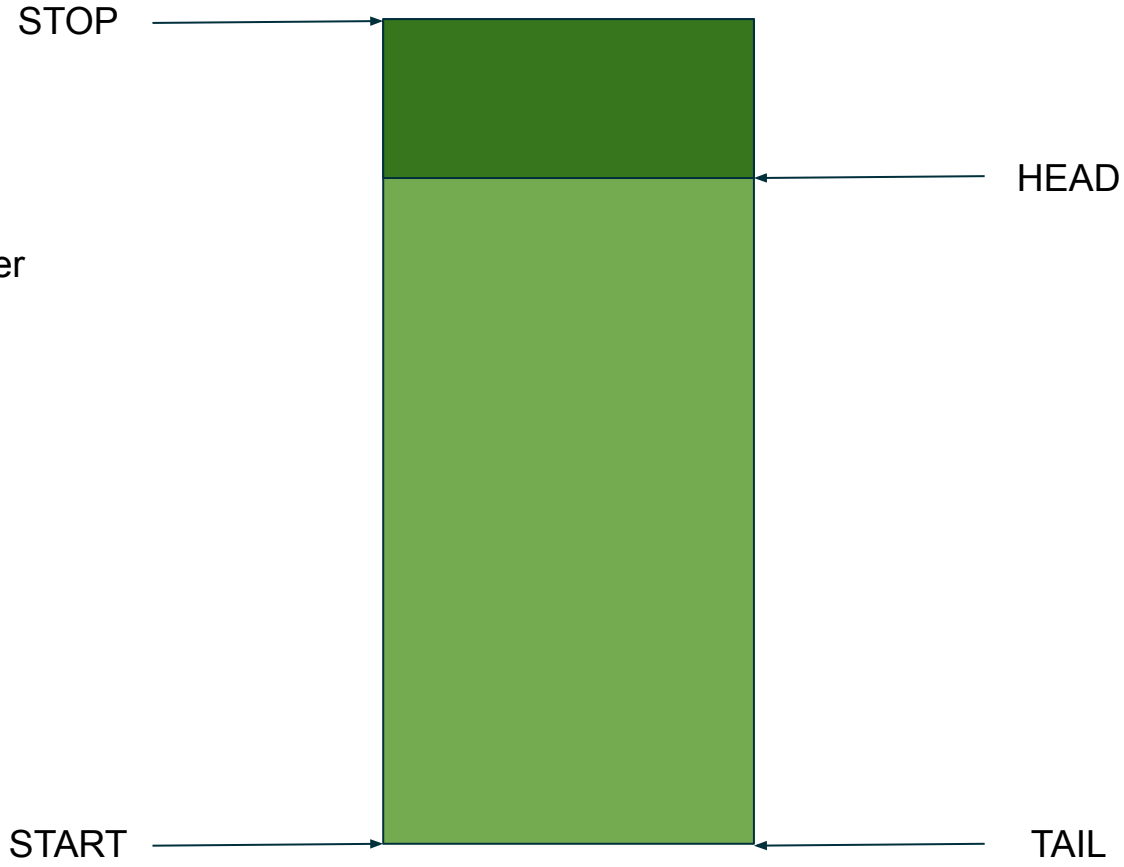
FPGA-side

Initial state



Dataserver-side

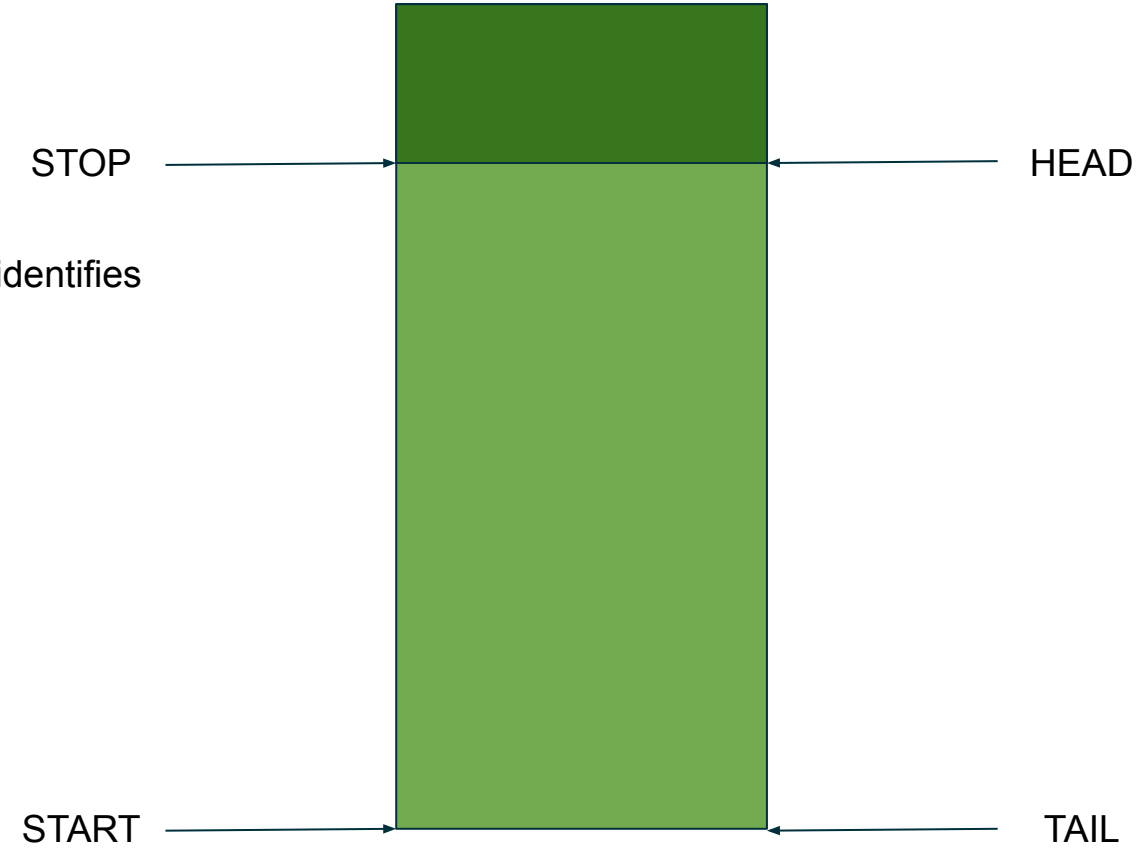
FPGA-side



FPGA puts some data into the buffer

Dataserver-side

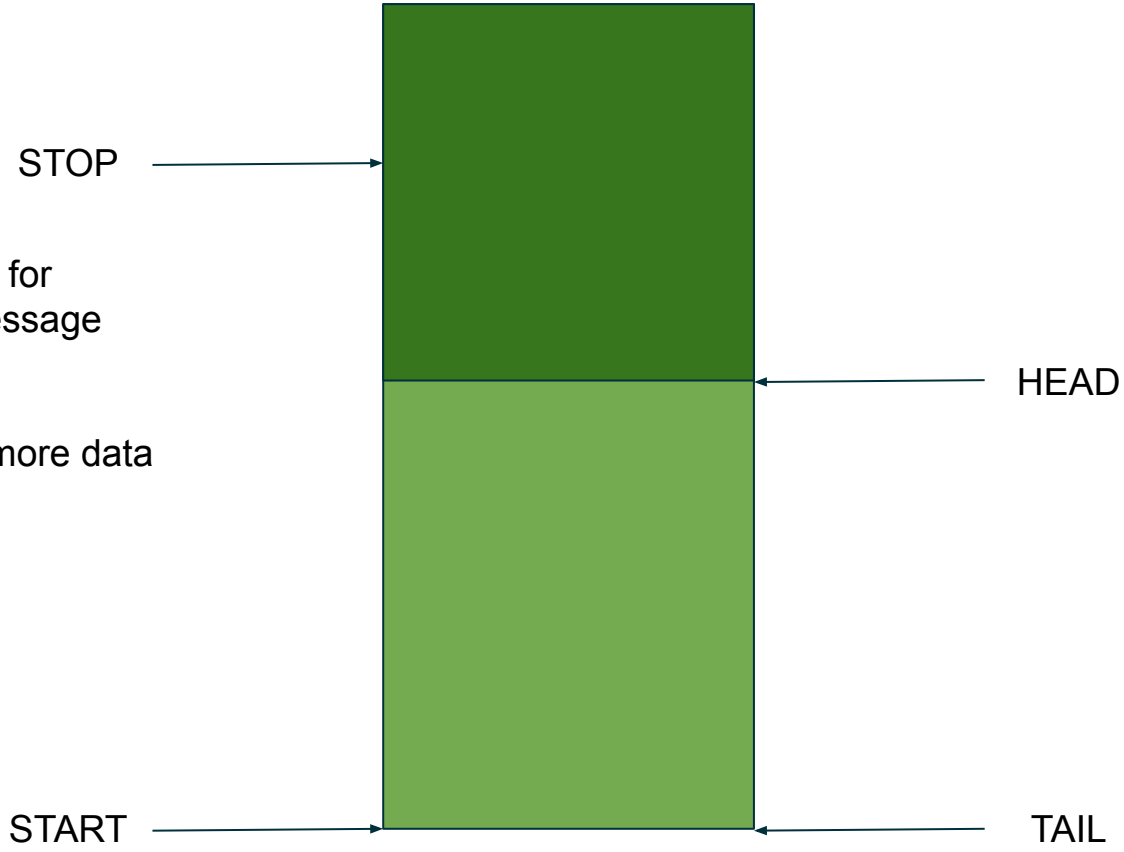
FPGA-side



The dataserver identifies new data

Dataserver-side

FPGA-side

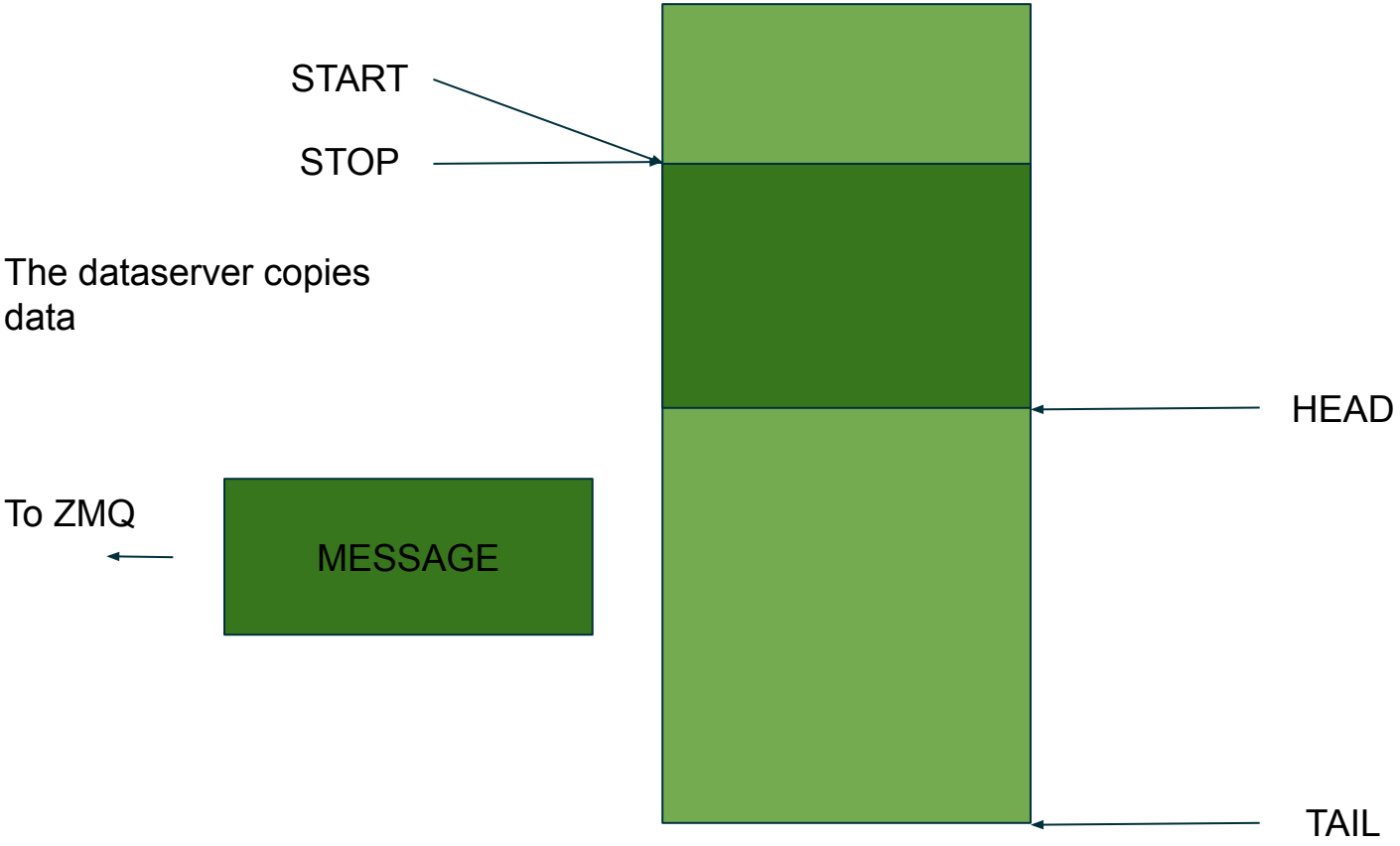


Dataserver waits for existing ZMQ message transfer to finish

The FPGA puts more data into the buffer

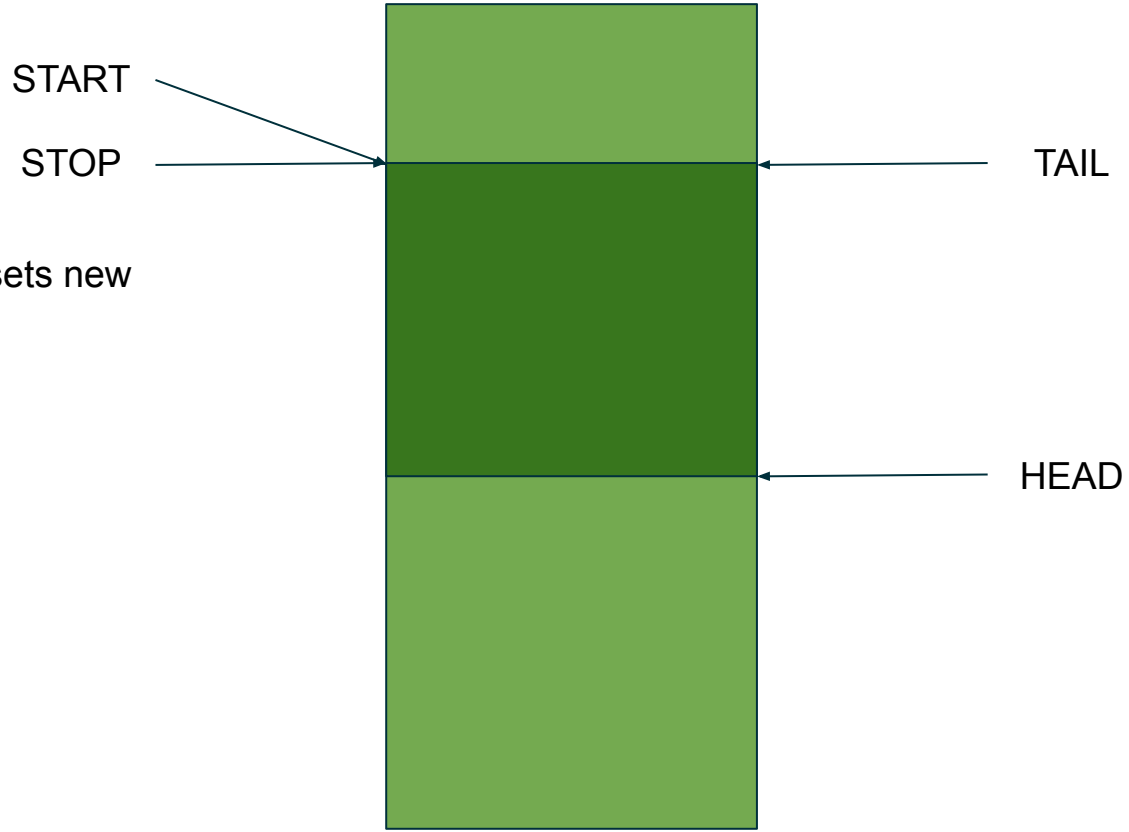
Dataserver-side

FPGA-side



Dataserver-side

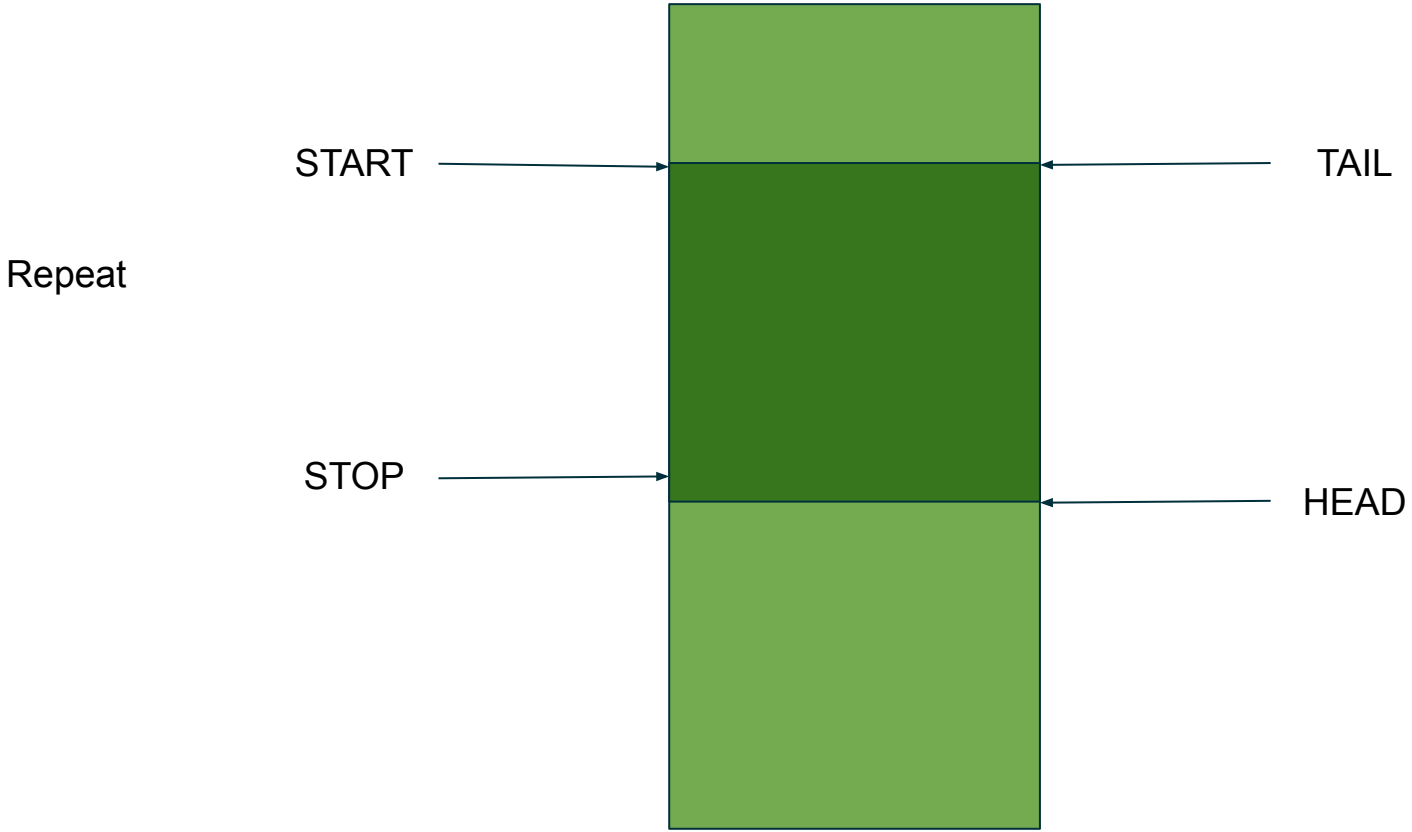
FPGA-side

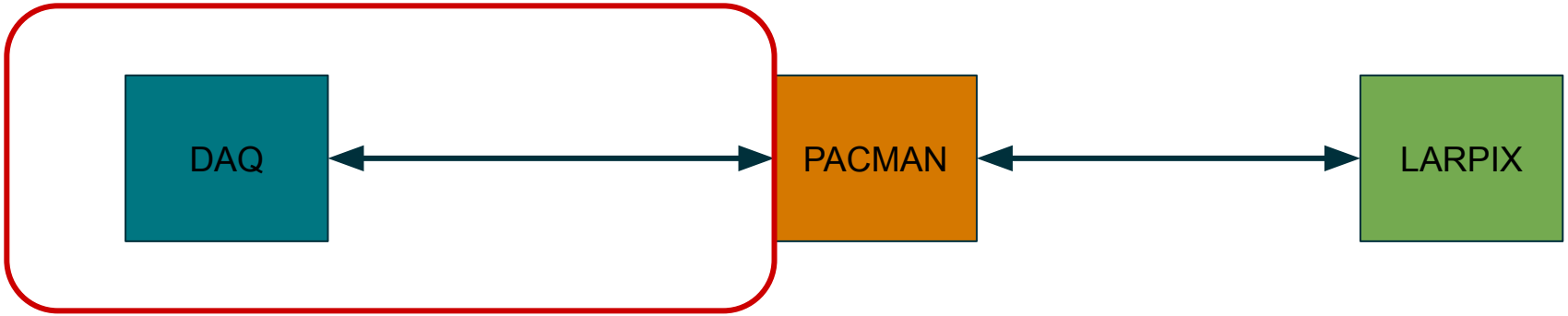


The dataserver sets new FPGA tail

Dataserver-side

FPGA-side




DAQ
PACMAN
LARPIX

larpix-control

At this point, no real “DAQ”, S Kohn had developed something for v1, but limited resources to develop this further for v2

if there is a group interested in this, I could help point the way on Sam’s work

larpix-control provides an interface to larpix with primary goals of

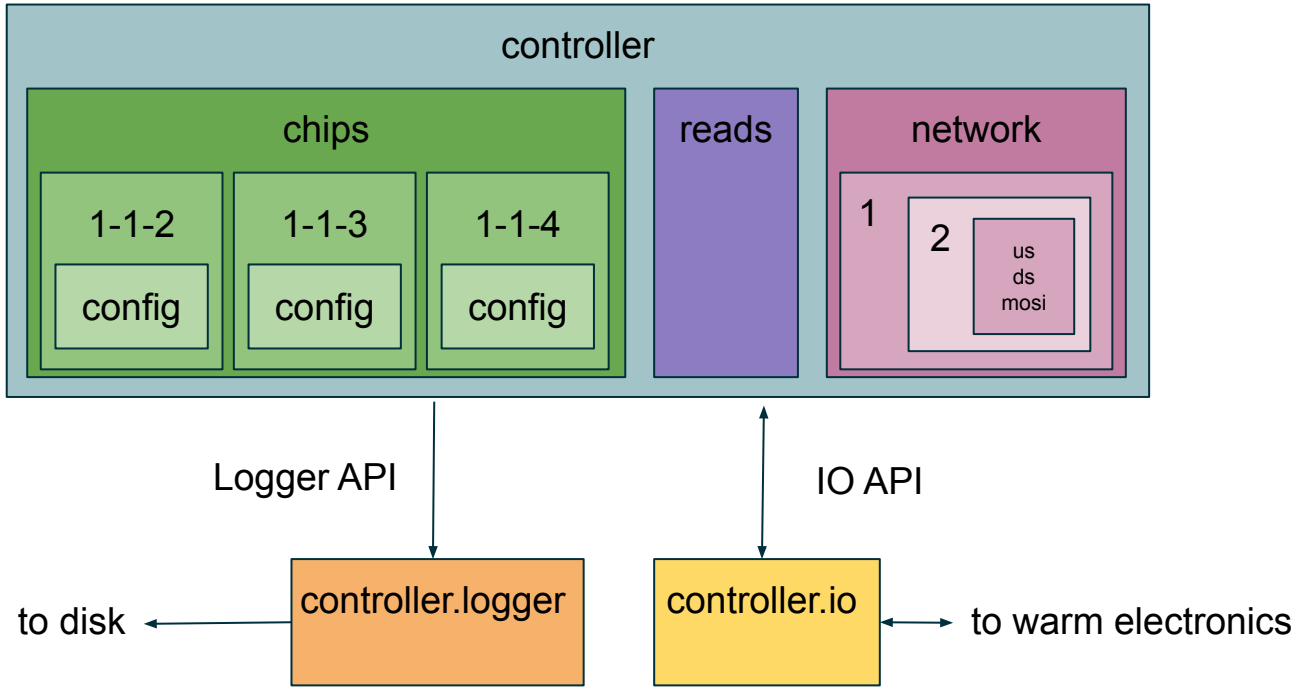
flexibility (easy scripting / extension)

on-the-fly system operation (interactive session)

good for R&D, less good for production

docs and a tutorial exists

Not the recommended approach for long term (ND), but capable of handling 2x2 data rates


controller
chips
1-1-2
config
1-1-3
config
1-1-4
config
reads
network
1
2
us
ds
mosi
Logger API
IO API
to disk
controller.logger
controller.io
to warm electronics

Controller

Typically, one overarching “Controller” per session / script

Core attributes

- controller.chips : representation of chip configurations
- controller.network : collection of directed graphs for hydra network configuration
- controller.reads : list of packets received from larpix
- controller.io : IO subclass object that handles data formatting and protocol
- controller.logger : Logger subclass object that handles data writing to disk

Dedicated functions for standard interactions

- controller.load() # load up a specific network structure
- controller.init_network() # send packets required to create hydra network
- controller.verify_configuration() # send config read packets and compare response to controller.chips configs
- controller.enforce_configuration() # verify_configuration, but send write config packets for any mis-match
- many others… see docs

controller.chips

Map of chip_key (‘<io_group>-<io_channel>-<chip_id>’) to Chip object

Chip class:

- chip.config : points to a Configuration subclass object
- other methods/attributes are rarely accessed by user directly, mostly just work with the chip configuration

Configuration subclasses:

- Subclassed on asic version
- Translates named configuration register to register address and bits
- Allows setting registers that actually span multiple- or sub-byte register addresses
  - chip.config.threshold_global = 26
  - chip.config.channel_mask = [0]*64
- Can update config registers via configuration file
  - chip.config.load(config_file)

controller.network

A hierarchy of networkx directed graphs to keep track of the desired hydra network

Not recommended to access directly, instead through

```
controller.add_chip() # creates chip object and updates network with new node
controller.add_network_link() # creates link between network nodes
controller.add_network_node() # adds new node only
controller.get_network_keys(<io_group>,<io_channel>) # traverses hydra network
```

Easy to get lost and confused when modifying the network manually

➔ *Generally recommend to use network config files to specify hydra network*

controller.reads

list of PacketCollection objects (list of Packets with some metadata)

Packet = unit of information from larpix / warm electronics

Contains all packets read from the controller.io, which can lead to large memory consumption for very long running scripts or high data rates

Clear with

controller.reads = []

controller.io

An IO subclass that handles translating larpix-control Packet objects into “messages” for the hardware and handles protocol for interfacing with warm electronics

Follows core API for base functions to allow modularity

- switching to a different warm electronics board messaging protocol just entails writing an IO subclass not all of larpix-control)

Note: some subclass-specific methods/attributes have been implemented but are not used by larpix-control

- handy for users, but maintains modularity for larpix-control

controller.io.pacman_io

Translates between PACMAN messages and Packets

header	->	TimestampPacket (unix timestamp)
data word	->	Packet_v2 (config read/write + data)
trigger word	->	TriggerPacket (PACMAN-level trigger timestamp)
sync word	->	SyncPacket (PACMAN-level clock timestamp)

PACMAN larpix-control docs

```
from larpix.io import PACMAN_IO
controller.io = PACMAN_IO(config_file) # config specifies io_group <-> ip address
controller.send(packets) ; controller.read() # standard IO API
controller.set_reg(0x0, 0x0) # PACMAN_IO only method to configure PACMAN
controller.get_reg(0x0) # PACMAN_IO only method to check PACMAN config
```

Extends to arbitrary number of PACMAN:

```
controller.set_reg(0x0, 0x0, io_group=3) # set config on specific PACMAN
controller.send(packets) # routing done via packet chip_key
```

controller.logger

A Logger subclass that handles translating larpix-control Packet objects into a persistent data format

Again follows a core API for base functions to allow modularity

switching to a different data format just requires writing a logger subclass

HDF5 logger:

```
from larpix.logger import HDF5Logger
controller.logger = HDF5Logger()
controller.logger.enable() # track all data sent/received by controller after this point
```

controller.logger

Unfortunately, for high data rates, the IO and logger classes can't keep up

The hack: Bypass packet interpretation and write messages directly to disk

```
controller.io.disable_packet_parsing = True # stop converting messages to Packets
controller.io.enable_raw_file_writing = True # write to raw data format
controller.read() # any new data goes to disk rather than controller.reads
```

Can keep disable_packet_parsing = False to access Packets from c.reads

a bit slower but not too much

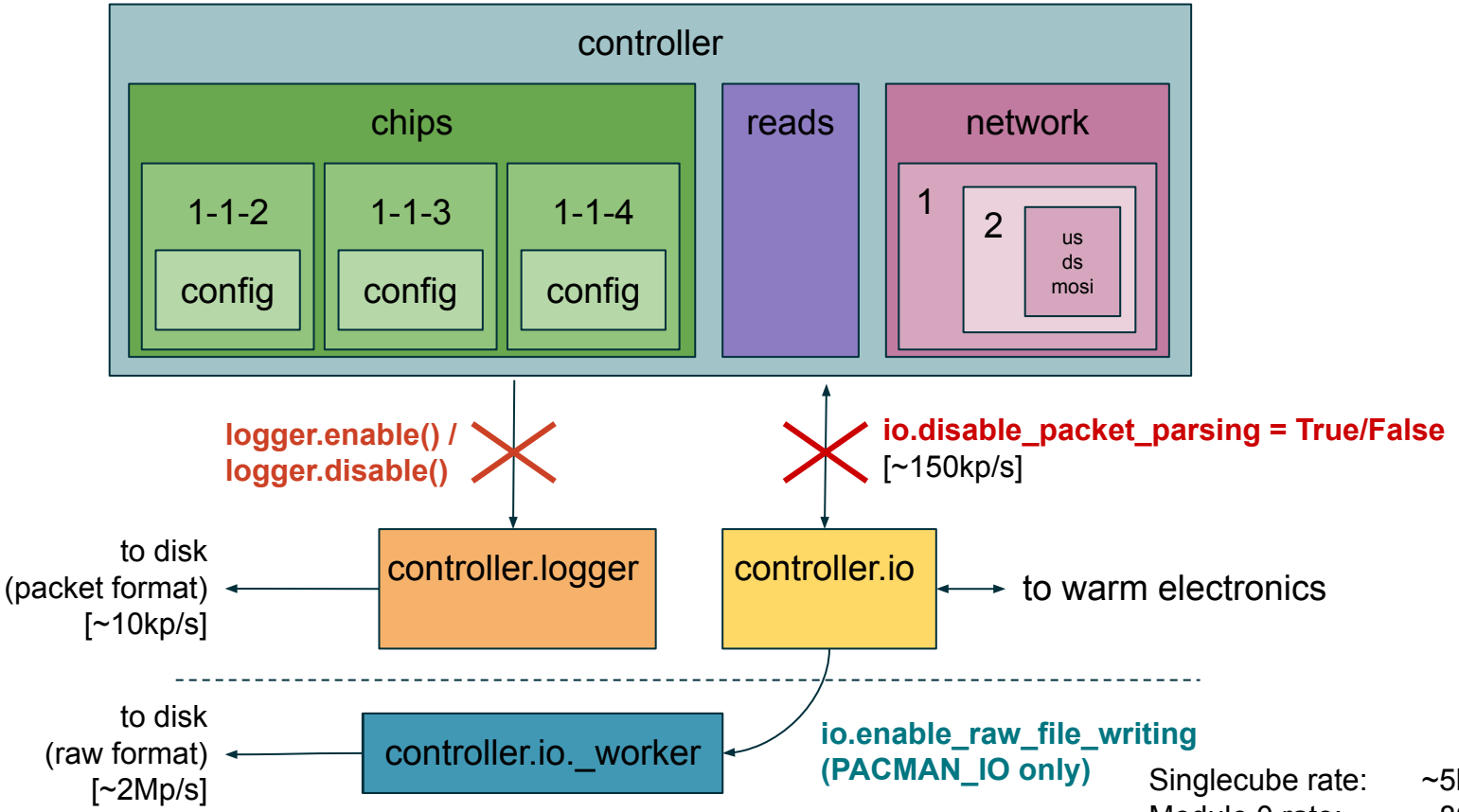
Script to convert between Packet-based HDF5 format and message-based HDF5 format:

```
scripts/convert_rawhdf5_to_hdf5.py -i <input file> -o <output file>
```

Raw HDF5 file opened in "single writer many reader" (SWMR) mode

Allows live monitoring of data files

Fast data writing hack



Singlecube rate:	~5kp/s
Module 0 rate:	~80kp/s
2x2 rate:	~320kp/s

Putting it all together

controller.load(filename) # creates Chip objects and creates controller.network links
controller.init_network(1,1) # brings up the desired Hydra network (io_group 1, io_channel 1)
controller.read_configuration(‘1-1-2’,0) # <- we’ll go in depth here

- Use chip and chip.config to create a list of appropriate config read Packets
- controller hands config Packets off to controller.io
- controller.io converts Packets into a bytestring message
- controller.io SUBscribes to ZMQ dataserver
- controller.io sends REQ message to PACMAN command server
- PACMAN command server puts each word into the shared memory buffer
- PACMAN command server starts DMA transfer
- Once finished, PACMAN replies REQ message to controller.io

Putting it all together

- PACMAN FPGA broadcasts each word to each UART TX block
- For the desired io channel, the UART TX block drives the UART to the first chip on the tile
- The first chip registers the packet, if chip id doesn’t match, places into own FIFO
- Upstream packet is grabbed from FIFO and passed to next chip according to upstream network config
- …
- Chip registers same chip id as in packet, put downstream packet with register value into own FIFO
- Packet handed chip-to-chip according to downstream network config
- PACMAN FPGA UART RX receives packet from first chip

Putting it all together

- Packet handed up through FPGA round-robins and FIFOs
- Packet reaches shared memory buffer
- PACMAN dataserver registers new data in buffer
- PACMAN dataserver formats a new message with packet
- PACMAN PUBlishes message to all SUBscribed devices (controller.io)
- controller requests new data from controller.io
- controller.io parses message into larpix-control Packets (if enabled)
- controller.io puts messages into raw data file worker queue (if enabled), restarts if necessary
- controller hands packets to controller.logger, keeps a copy in controller.reads
- controller.logger writes packets to file (if enabled)
- controller.io unSUBscribes from PACMAN dataserver

Ready for next command

List of useful/relevant links

larpix-v2 datasheet: link

larpix-control docs: link
larpix-control code: link

PACMAN interface doc: link
PACMAN server code: link